

**SÁCH  
TỰ HỌC**

**DƯƠNG QUANG THIÊN**  
biên soạn

**.NET Toàn tập**

**Tập 2**

# **C# và .NET Framework**

**Lập trình Visual C# thế nào?**

**C# căn bản**



**C# và  
.NET Framework**



**GUI và user control**



**ADO .NET**



**ASP .NET**



**Crystal Report**

**NHÀ XUẤT BẢN TỔNG HỢP TP.HCM**

# C#

## và .NET Framework

**Lập trình Visual C# thế nào?**

DƯƠNG QUANG THIÊN  
biên soạn

**.NET Toàn tập**

---

**Tập 2**

# **C# và .NET Framework**

**Lập trình Visual C# thế nào?**

NHÀ XUẤT BẢN TỔNG HỢP TP. HCM



# Mục Lục

<b>LỜI MỞ ĐẦU .....</b>	<b>17</b>
-------------------------	-----------

## **Chương 1**                      ***Xuất Nhập Dữ Liệu & Sản Sinh Hàng Loạt Đối Tượng***

<b>1.1. Các tập tin và Thư mục .....</b>	<b>28</b>
1.1.1 Khảo sát namespace System.IO .....	28
1.1.2 Các lớp Directory(Info) và File(Info) .....	29
1.1.2.1 Lớp cơ bản <i>abstract FileSystemInfo</i> .....	30
1.1.2.2 Làm việc với lớp <i>DirectoryInfo</i> .....	31
1.1.2.3 Tạo một đối tượng <i>DirectoryInfo</i> .....	31
1.1.2.4 Enumeration <i>FileAttributes</i> .....	33
1.1.3 Rảo xem các tập tin thông qua lớp <i>DirectoryInfo</i> .....	34
1.1.4 Rảo xem các thư mục con .....	35
1.1.5 Tạo thư mục con thông qua lớp <i>DirectoryInfo</i> .....	38
1.1.6 Các thành viên static của lớp <i>Directory</i> .....	40
1.1.7 Làm việc với các tập tin .....	41
1.1.7.1 Rảo xem các tập tin và thư mục .....	43
1.1.7.2 Tạo một tập tin vật lý .....	38
1.1.7.3 Khảo sát hàm hành sự <i>FileInfo.Open()</i> .....	45
1.1.7.4 Các hàm <i>FileInfo.OpenRead()</i> và <i>FileInfo.OpenWrite()</i> .....	46
1.1.7.5 Các hàm <i>OpenText()</i> , <i>CreateText()</i> và <i>AppendText()</i> của <i>FileInfo</i> .....	49
1.1.7.6 Thay đổi các tập tin .....	50
<b>1.2 Đọc và viết dữ liệu .....</b>	<b>53</b>
1.2.1 Làm việc với các tập tin nhị phân .....	54
1.2.2 Làm việc với <i>FileStream</i> .....	56
1.2.3 Làm việc với <i>MemoryStream</i> .....	57
1.2.4 Làm việc với <i>BufferedStream</i> .....	59
1.2.5 Làm việc với những tập tin văn bản .....	60
1.2.5.1 <i>Viết và Đọc một tập tin văn bản</i> .....	62
1.2.6 Làm việc với <i>StringWriter</i> và <i>StringReader</i> .....	66
1.2.7 Làm việc với dữ liệu nhị phân(các lớp <i>BinaryReader</i> & <i>BinaryWriter</i> ) .....	69
<b>1.3 Xuất nhập dữ liệu bất đồng bộ(Asynchronous I/O) .....</b>	<b>72</b>
<b>1.4 Xuất nhập dữ liệu trên mạng (Network I/O) .....</b>	<b>76</b>
1.4.1 Tạo một Network Streaming Server .....	78

1.4.2 Tạo một Streaming Network Client.....	80
1.4.3 Thụ lý nhiều kết nối .....	82
1.4.4 Asynchronous Network File Streaming .....	85
<b>1.5. Sàn sinh hàng loạt (Serialization) .....</b>	<b>92</b>
1.5.1 Vai trò của Object Graph.....	93
1.5.2 Cấu hình các đối tượng cho việc sàn sinh hàng loạt .....	93
1.5.2.1 Sử dụng một Formatter.....	96
1.5.2.2 Vai trò của namespace System.Runtime.Serialization .....	97
1.5.3 Sàn sinh hàng loạt sử dụng đến Binary Formatter .....	98
1.5.3.1 Cho serialize đối tượng .....	99
1.5.3.2 Cho deserialize đối tượng .....	100
1.5.4 Sàn sinh hàng loạt sử dụng đến SOAP Formatter .....	102
<b>1.6 Thụ lý dữ liệu vãng lai .....</b>	<b>103</b>
<b>1.7 Isolated Storage.....</b>	<b>106</b>
<b>1.8 Custom Serialization và giao diện ISerializable .....</b>	<b>109</b>
1.8.1 Một thí dụ đơn giản .....	111
<b>1.9 Một ứng dụng Windows Forms: Car Logger .....</b>	<b>113</b>

## Chương 2      *Xây dựng một ứng dụng Windows*

<b>2.1 Tổng quan về namespace Windows.Forms .....</b>	<b>124</b>
<b>2.2 Tương tác với các lớp Windows Forms .....</b>	<b>126</b>
2.2.2 Tạo bằng tay một cửa sổ chính .....	127
2.2.3 Tạo một dự án Windows Form trên Visual Studio .NET .....	129
<b>2.3 Tìm hiểu lớp System.Windows.Forms.Application .....</b>	<b>133</b>
2.3.1 Thử thực hành với lớp Application .....	135
2.3.2 Phản ứng trước tình huống ApplicationExit .....	136
2.3.3 Xử lý trước các thông điệp với lớp Application .....	137
<b>2.4 Hình thù một biểu mẫu ra sao?.....</b>	<b>138</b>
2.4.1 Lớp Object, MarshalByRefObject .....	139
2.4.2 Lớp Component .....	140
2.4.3 Lớp Control .....	142

2.4.3.1 Cho đặt để style của một biểu mẫu.....	144
2.4.4 Điều khiển các tình huống.....	147
2.4.4.1 Đáp ứng tình huống MouseUp - Phần 1.....	150
2.4.4.2 Xác định xem nút nào trên con chuột bị click .....	152
2.4.4.3 Đáp ứng các tình huống Mouse Events - Phần 2 .....	152
2.4.5 Đáp ứng các tình huống Keyboard.....	153
2.4.6 Lớp Control - Phần chót .....	155
2.4.6.1 Căn bản về tô vẽ (painting) .....	158
2.4.7 Lớp ScrollableControl.....	158
2.4.8 Lớp ContainerControl.....	159
2.4.9 Lớp Form .....	160
2.4.9.1 Thử chơi một tí với lớp Form .....	161
<b>2.5 Tạo những trình đơn với Windows Forms .....</b>	<b>163</b>
2.5.1 Lớp Menu\$MenuItemCollection .....	165
<b>2.6 Tạo một hệ thống trình đơn.....</b>	<b>166</b>
2.6.1 Thêm một mục chọn trình đơn chớp bu khác .....	168
2.6.2 Tạo một trình đơn shortcut.....	169
2.6.3 Tô điểm thêm hệ thống trình đơn của bạn .....	171
2.6.4 Tạo một trình đơn dùng Visual Studio .NET IDE .....	174
<b>2.7 Tìm hiểu thanh tình trạng và lớp StatusBar .....</b>	<b>176</b>
2.7.1 Cơ bản về Status Bar.....	176
2.7.2 Thử tạo một thanh tình trạng .....	178
2.7.3 Đồng bộ hoá thanh tình trạng với một trình đơn .....	180
2.7.4 Làm việc với lớp Timer.....	181
2.7.5 Cho hiển thị dòng nhắc đối với mục chọn trình đơn .....	182
<b>2.8 Tạo một thanh công cụ .....</b>	<b>183</b>
2.8.1 Thêm hình ảnh lên các nút thanh công cụ .....	188
2.8.2 Tạo thanh công cụ vào lúc thiết kế.....	189
2.8.2.1 Thêm một ImageList vào lúc thiết kế .....	190
2.8.3 Đồng bộ hóa thanh công cụ.....	192
<b>2.9 Một ứng dụng Windows Forms tối thiểu và trọn vẹn .....</b>	<b>195</b>
<b>2.10 Xây dựng một ứng dụng Windows Form trọn vẹn .....</b>	<b>197</b>
2.10.1 Tạo biểu mẫu giao diện cơ bản .....	198
2.10.2 Cho điền các TreeView Control.....	200
2.10.2.1 Các đối tượng TreeNode .....	200
2.10.2.2 Rào đê quy xuyên qua các thư mục con.....	202
2.10.2.3 Đi lấy các tập tin trên thư mục .....	203
2.10.3 Thụ lý các tình huống của TreeView.....	203
2.10.3.1 Click TreeView nguồn .....	204
2.10.3.2 Click TreeView đích .....	205

2.10.4	Thụ lý tình huống Click nút <Clear> .....	206
2.10.5	Thi công tình huống Click nút <Copy> .....	206
2.10.5.1	Đi lấy các tập tin được tuyển chọn .....	206
2.10.5.2	Cho sắp xếp các tập tin được chọn ra .....	208
2.10.6	Thi công tình huống Click nút <Delete> .....	210
<b>2.11</b>	<b>Tương tác với System Registry .....</b>	<b>215</b>
2.11.1	Registry .....	216
2.11.2	Các lớp .NET Registry .....	218
<b>2.12</b>	<b>Tương tác với Event Viewer .....</b>	<b>223</b>

## Chương 3      *Tìm hiểu về Assembly và cơ chế Version*

<b>3.1</b>	<b>Tổng quan về .NET Assembly.....</b>	<b>229</b>
3.1.1	Các tập tin PE.....	230
3.1.2	Cấu trúc của một Assembly .....	230
3.1.3	Metadata là gì?.....	232
3.1.4	Assembly Manifest .....	232
3.1.4.1	Các module trên Manifest .....	233
3.1.4.2	Module Manifest .....	234
3.1.4.3	Các assembly khác được cần đến.....	235
3.1.5	Hai cái nhìn của một assembly: vật lý và logic.....	235
3.1.6	Assembly khuyến khích việc tái sử dụng đoạn mã .....	236
3.1.7	Assembly thiết lập Type Boundary và Security Boundary .....	237
3.1.8	Phiên bản hóa một assembly .....	237
<b>3.2</b>	<b>Thử xây dựng một Single File Test Assembly.....</b>	<b>238</b>
3.2.1	Một ứng dụng client C#: CSharpCarClient .....	241
3.2.2	Một ứng dụng client VB.NET: VBCarClient .....	243
3.2.3	Kế thừa xuyên ngôn ngữ lập trình .....	244
3.2.4	Thử khảo sát manifest của CarLibrary.dll .....	245
3.2.5	Khảo sát các kiểu dữ liệu của CarLibrary.dll.....	248
<b>3.3</b>	<b>Thử khảo sát Multi-Module Assembly .....</b>	<b>250</b>
3.3.1	Những lợi điểm khi dùng multi-module assembly .....	251
3.3.2	Xây dựng một multi-module assembly .....	252
<b>3.4</b>	<b>Thử tìm hiểu Private Assembly .....</b>	<b>259</b>
3.4.1	Cơ bản về Probing.....	261
3.4.1.1	Nhận diện một Private Assembly .....	261



3.4.1.2 Private Assembly và các tập tin XML Configuration .....	262
3.4.1.3 Những đặc thù trong việc gắn kết với một private assembly...	263
<b>3.5 Thử tìm hiểu Shared Assembly .....</b>	<b>265</b>
3.5.1 Kết thúc cơ chế ác mộng DLL .....	266
3.5.2 Tìm hiểu cơ chế về phiên bản .....	267
3.5.3 Tìm hiểu về Strong Name .....	268
3.5.4 Xây dựng một shared assembly .....	269
3.5.5 Cài đặt assembly vào GAC .....	271
3.5.6 Sử dụng một Shared Assembly .....	272
3.5.7 Ghi nhận thông tin về Version .....	274
3.5.7.1 "Đồng cứng" SharedAssembly hiện hành .....	274
3.5.7.2 Xây dựng SharedAssembly, Version 2.0 .....	276
3.5.7.3 Tìm hiểu cơ chế kết nối mặc nhiên với phiên bản .....	277
3.5.8 Khai báo cơ chế phiên bản custom .....	278

## **Chương 4**      ***Tìm hiểu về Type Attributes và Reflection***

<b>4.1 Tìm hiểu về Attributes .....</b>	<b>279</b>
4.1.1 Attribute "bẩm sinh" (intrinsic attribute) .....	280
4.1.1.1 Attribute Target .....	280
4.1.1.2 Áp dụng các attribute.....	281
4.1.2 Custom Attribute .....	283
4.1.2.1 Khai báo một attribute .....	284
4.1.2.2 Đặt tên cho một attribute .....	284
4.1.2.3 Xây dựng một attribute .....	285
4.1.2.4 Sử dụng một attribute.....	285
4.1.3 Assembly Level Attributes và Module Level Attributes .....	289
4.1.4 Tập tin Studio.NET AssemblyInfo.cs .....	290
4.1.5 Khám phá các attributes vào lúc chạy.....	291
<b>4.2 Tìm hiểu về Reflection .....</b>	<b>292</b>
4.2.1 Type Class .....	293
4.2.1.1 Nhận về một đối tượng Type.....	293
4.2.1.2 Sử dụng lớp Type .....	294
4.2.2 Nhìn xem Metadata .....	298
4.2.2.1 Khảo sát namespace System.Reflection.....	298
4.2.2.2 Sử dụng System.Reflection.MemberInfo .....	298
4.2.3 Phát hiện các kiểu dữ liệu .....	299
4.2.3.1 Nạp một assembly.....	300
4.2.3.2 Liệt kê các kiểu dữ liệu trên một assembly được qui chiếu .....	302
4.2.4 Phản chiếu trên một kiểu dữ liệu.....	303
4.2.4.1 Liệt kê các thành viên của một lớp.....	303

4.2.4.2	<i>Truy tìm các thành viên đặc biệt của kiểu dữ liệu</i>	304
4.2.4.3	<i>Liệt kê các hàm hành sự hoặc các thông số của một hàm hành sự</i>	306
4.2.5	<i>Tìm hiểu triệu gọi động trễ (late binding)</i>	308
4.2.5.1	<i>Lớp Activator</i>	308
<b>4.3</b>	<b>Tìm hiểu và Xây dựng Dynamic Assembly</b>	<b>311</b>
4.3.1	<i>Tìm hiểu namespace System.Reflection.Emit</i>	311
4.3.2	<i>Emitting một Dynamic Assembly</i>	312
4.3.3	<i>Sử dụng một Dynamic Assembly được kết sinh thể nào</i>	316
4.3.4	<i>Reflection Emit</i>	318
4.3.4.1	<i>Dynamic Invocation sử dụng đến InvokeMember()</i>	321
4.3.4.2	<i>Dynamic Invocation sử dụng Interfaces</i>	330
4.3.4.3	<i>Dynamic Invocation sử dụng Reflection Emit</i>	335

## Chương 5 *Marshaling và Remoting*

<b>5.1</b>	<b>Application Domains</b>	<b>343</b>
5.1.1	<i>Chơi một chút với AppDomain</i>	345
5.1.2	<i>Một thí dụ sử dụng AppDomain</i>	347
5.1.2.1	<i>Tạo và Sử dụng App Domain</i>	347
5.1.2.2	<i>Marshalling xuyên ranh giới App Domain</i>	349
<b>5.2</b>	<b>Phạm trù (context)</b>	<b>355</b>
5.2.1	<i>Các đối tượng contex-bound và context-agile</i>	356
5.2.2	<i>Cho marshal xuyên biên giới phạm trù</i>	357
<b>5.3</b>	<b>Remoting</b>	<b>358</b>
5.3.1	<i>Tìm hiểu kiểu dữ liệu đối tượng Server</i>	358
5.3.2	<i>Khai báo một Server với một Interface</i>	359
5.3.3	<i>Xây dựng một Server</i>	359
5.3.4	<i>Xây dựng một Client</i>	362
5.3.5	<i>Sử dụng SingleCall</i>	365
5.3.6	<i>Tìm hiểu RegisterWellKnownServiceType</i>	365
5.3.7	<i>Tìm hiểu Endpoints</i>	366

## Chương 6 *Mạch Trình và Đồng Bộ Hóa*

<b>6.1</b>	<b>Các mạch trình</b>	<b>371</b>
6.1.1	<i>Khảo sát namespace System.Threading</i>	371
6.1.1.1	<i>Thử khảo sát lớp Thread</i>	372
6.1.2	<i>Khởi động các mạch trình</i>	373

6.1.3 Đặt tên thân thiện cho mạch trình.....	376
6.1.4 Ráp lại các mạch trình .....	377
6.1.5 Cho treo lại các mạch trình .....	378
6.1.6 Cho khai tử mạch trình .....	378
<b>6.2 Đồng bộ hóa (synchronization) .....</b>	<b>382</b>
6.2.1 Sử dụng từ chốt C# Locks.....	387
6.2.2 Sử dụng System.Threading.Monitor.....	388
<b>6.3 Điều kiện chạy đua và hiện tượng chết chùm (deadlock).....</b>	<b>393</b>
6.3.1 Các điều kiện chạy đua .....	394
6.3.2 Chết chùm (deadlock) .....	394

## Chương 7 *Tương tác với Unmanaged code*

<b>7.1 Tìm hiểu vấn đề Interoperability .....</b>	<b>396</b>
<b>7.2 Tìm hiểu về Namespace System.Runtime.InteropServices .....</b>	<b>398</b>
<b>7.3 Tương tác với .DLL viết theo C#.....</b>	<b>399</b>
7.3.1 Khai báo vùng mục tin ExactSpelling .....	401
7.3.2 Khai báo vùng mục tin CharSet .....	402
7.3.3 Khai báo vùng mục tin CallingConventions và EntryPoint .....	402
<b>7.4 Tìm hiểu .NET to COM Interoperability .....</b>	<b>404</b>
7.4.1 Cho trưng COM Type như là .NET tương đương .....	405
7.4.2 Quản lý cái đếm qui chiếu của một CoClass.....	406
7.4.3 Che dấu Low-Level COM Interfaces .....	406
7.4.4 Tạo một COM server rất đơn giản viết theo Visual Basic.....	407
7.4.4.1 Quan sát IDL được kết sinh đối với Visual Basic COM Server ..	410
7.4.5 Xây dựng một Visual Basic COM Client đơn giản.....	411
7.4.6 Nhập khẩu Type Library .....	412
7.4.7 Qui chiếu tập tin SimpleAssembly.dll .....	413
7.4.7.1 Gắn kết sớm về lớp CoCalc COM.....	414
7.4.7.2 Gắn kết sớm sử dụng Visual Studio .NET .....	415
7.4.8 Gắn kết trễ về coclass CoCalc.....	415
7.4.9 Khảo sát assembly được kết sinh .....	419
<b>7.5 Sử dụng ActiveX control trên .NET.....</b>	<b>421</b>
7.5.1 Tạo một ActiveX control.....	422
7.5.2 Nhập khẩu một ActiveX control vào .NET .....	425
7.5.2.1 Nhập khẩu một ô control .....	425
7.5.2.2 Thêm một ô control vào Visual Studio toolbox.....	427

<b>7.6 Xây dựng một ATL Test Server .....</b>	<b>428</b>
7.6.1 “Thêm mắm thêm muối” cho [default] COM Interface .....	429
7.6.2 Cho phát pháo một COM Event .....	431
7.6.3 Cho tung ra một COM Error .....	433
7.6.4 Cho trưng ra một subobject nội bộ (và sử dụng SAFEARRAY) .....	434
7.6.5 Bước cuối cùng: cấu hình hóa một IDL Enumeration .....	436
7.6.6 Thử tạo một Visual Basic 6.0 Test Client .....	437
7.6.7 Xây dựng Assembly và xem xét tiến trình chuyển đổi .....	439
7.6.7.1 <i>Type Library Conversion</i> .....	440
7.6.7.2 <i>COM Interface Conversion</i> .....	440
7.6.7.3 <i>Parameter Attribute Conversion</i> .....	441
7.6.7.4 <i>Interface Hierarchy Conversion</i> .....	442
7.6.7.5 <i>Coclass (và COM Properties) Conversion</i> .....	443
7.6.7.6 <i>COM Enumeration Conversion</i> .....	446
7.6.7.7 <i>Làm việc với COM SAFEARRAY</i> .....	447
7.6.7.8 <i>Chận hứng các tình huống COM</i> .....	448
7.6.7.9 <i>Thụ lý COM Error</i> .....	450
7.6.8 Toàn bộ ứng dụng C# Client .....	451
<b>7.7 Tìm hiểu Interoperability từ COM chuyển qua .NET .....</b>	<b>452</b>
7.7.1 Vai trò của CCW (COM Callable Wrapper) .....	453
7.7.2 Tìm hiểu Class Interface .....	454
7.7.2.1 Định nghĩa một Class Interface .....	455
7.7.3 Xây dựng .NET type của bạn .....	455
7.7.4 Kết sinh Type Library và cho đăng ký .Net Type .....	456
7.7.5 Quan sát Exported Type Information .....	456
7.7.5.1 <i>_Object Interface</i> .....	458
7.7.5.2 <i>Câu lệnh Library được kết sinh</i> .....	458
7.7.6 Nhìn xem kiểu dữ liệu sử dụng OLE/COM Object Viewer .....	459
7.7.7 Quan sát các mục vào Registry .....	460
7.7.8 Xây dựng một Visual Basic 6.0 Test Client .....	462
7.7.9 Các vấn đề do .NET-to-COM đặt ra .....	463
7.7.9.1 <i>Quan sát thông tin kiểu dữ liệu của lớp BaseClass</i> .....	464
7.7.9.2 <i>Quan sát thông tin kiểu dữ liệu của lớp DerivedClass</i> .....	465
<b>7.8 Điều khiển phần Generated IDL (hoặc ảnh hưởng lên TlbExp.exe)</b>	<b>466</b>
7.8.1 Quan sát thông tin kiểu dữ liệu COM được kết sinh (IDL) .....	467
7.8.2 Tương tác với Assembly Registration .....	468
<b>7.9 Tương tác với COM+ Services .....</b>	<b>469</b>
<b>7.10 Xây dựng các kiểu dữ liệu chịu chơi COM+ .....</b>	<b>472</b>
7.10.1 Xây dựng một COM+ Aware C# Type .....	472

7.10.1.1 Thêm các attribute thiên COM+ cấp assembly .....	474
7.10.1.2 Cấu hình hóa assembly vào COM+ Catalog .....	475
7.10.2 Quan sát Component Service Explorer .....	476

## **Chương 8**      *Lập trình ứng dụng Web với Web Forms và ASP .NET*

<b>8.1 Phân biệt Web Application và Web Serser .....</b>	<b>479</b>
8.1.1 Tìm hiểu Virtual Directories .....	480
<b>8.2 Tìm hiểu về Web Forms .....</b>	<b>482</b>
8.2.1 Các tình huống trên Web Form .....	483
8.2.1.1 Các tình huống postback so với non-postback .....	484
8.2.1.2 Tình trạng châu làm việc .....	484
8.2.2 Chu kỳ sống của Web Form .....	484
<b>8.3 Cấu trúc cơ bản của một tài liệu HTML.....</b>	<b>486</b>
8.3.1 Cơ bản về định dạng văn bản trên HTML .....	488
8.3.2 Làm việc với Format Headers .....	490
8.3.3 Visual Studio.NET HTML Editors .....	491
<b>8.4 Triển khai HTML Form .....</b>	<b>492</b>
8.4.1 Xây dựng User Interface .....	494
8.4.2 Thêm vào một hình ảnh.....	496
<b>8.5 Vai trò kịch bản phía Client .....</b>	<b>497</b>
8.5.1 Một thí dụ về Client-Side Scripting .....	498
8.5.2 Kiểm tra hợp lệ đối với trang HTML Page1.htm .....	500
<b>8.6 Trình duyệt dữ liệu biểu mẫu (GET &amp; POST) .....</b>	<b>502</b>
8.6.1 Phân tích ngữ nghĩa của một Query String .....	503
<b>8.7 Tạo một Active Server Page cổ điển .....</b>	<b>504</b>
8.7.1 Đáp lại POST Submissions.....	506
<b>8.8 Xây dựng một ứng dụng ASP.NET .....</b>	<b>508</b>
<b>8.9 Vài vấn đề đối với ASP cổ điển.....</b>	<b>509</b>
8.9.1 Một số lợi điểm của ASP.NET .....	509
<b>8.10 ASP.NET Namespaces .....</b>	<b>510</b>
8.10.1 Các lớp cốt lõi của System.Web .....	510

<b>8.11 Tìm hiểu sự phân biệt Application/Session.....</b>	<b>511</b>
<b>8.12 Tạo một C# Web Application đơn giản.....</b>	<b>513</b>
8.12.1 Thử khảo sát tập tin *.aspx ban đầu.....	515
8.12.2 Thử khảo sát tập tin Web .Config .....	516
8.12.3 Thử khảo sát tập tin Global.asax .....	516
8.12.4 Thêm vài đoạn mã đơn giản C# .....	517
<b>8.13 Thử xem kiến trúc của một ứng dụng ASP.NET Web .....</b>	<b>518</b>
8.13.1 Kiểu dữ liệu System.Web.UI.Page .....	519
8.13.2 *.aspx/Codebehind Connection .....	520
8.13.3 Làm việc với Page.Request Property .....	521
8.13.4 Làm việc với thuộc tính Page.Response .....	523
8.13.5 Làm việc với thuộc tính Page.Application .....	524
<b>8.14 Gỡ rối và Lăn theo dấu vết trên các ứng dụng ASP.NET Web ..</b>	<b>525</b>
<b>8.15 Tìm hiểu các ô WebForm Controls .....</b>	<b>528</b>
<b>8.16 Làm việc với các ô WebForm Controls .....</b>	<b>530</b>
8.16.1 Dẫn xuất các ô WebForm Controls .....	532
<b>8.17 Các loại ô WebForm Controls.....</b>	<b>533</b>
8.17.1 Làm việc với những ô control WebForm “bẩm sinh” .....	533
8.17.1.1 Tạo một nhóm ô Radio Buttons.....	535
8.17.1.2 Tạo một ô TextBox nhiều hàng và rảo xem được.....	536
8.17.2 Các ô Rich Controls .....	536
8.17.2.1 Làm việc với Calendar Control .....	537
8.17.2.2 Làm việc với AdRotator.....	539
8.17.3 Các ô control chuyên về căn cứ dữ liệu (Datacentric Control) ....	541
8.17.3.1 Cho điền dữ liệu vào một DataGrid.....	541
8.17.3.2 Đôi điều về Data Binding .....	543
8.17.4 Các ô control kiểm tra hợp lệ dữ liệu (Validation Controls).....	544
8.17.5 Thụ lý tình huống WebForms Control .....	546
<b>8.18 Một ứng dụng Web Form .....</b>	<b>548</b>
8.18.1 Kết nối với căn cứ dữ liệu .....	550
8.18.2 Thụ lý tình huống Postback .....	553

<b>9.1 Tìm hiểu vai trò của Web Services.....</b>	<b>556</b>
<b>9.2 Các thành phần cốt lõi của Web Service .....</b>	<b>557</b>
9.2.1 Wire Protocol là gì? .....	558
9.2.2 Web Service Description Services là gì? .....	558
9.2.3 Discovery Services là gì?.....	558
<b>9.3 Tìm hiểu Namespace Web Service .....</b>	<b>559</b>
9.3.1 Thử xét namespace System.Web.Services.....	559
9.3.1.1 Xây dựng một Web Service đơn giản.....	560
9.3.1.2 Lớp WebMethodAttribute .....	564
9.3.1.3 Lớp cơ bản System.Web.Services.WebService .....	568
9.3.2 Thử tìm hiểu namespace System.Web.Services.Description .....	569
9.3.3 Thử xét namespace System.Web.Services.Protocols .....	571
9.3.3.1 Chuyển tin sử dụng nghi thức HTTP GET và HTTP POST .....	572
9.3.3.2 Chuyển tin sử dụng nghi thức SOAP.....	574
<b>9.4 WSDL trong đoạn mã C# - Kết sinh một proxy .....</b>	<b>575</b>
9.4.1 Xây dựng một proxy sử dụng wsdl.exe .....	576
9.4.1.1 Quan sát Proxy Code .....	576
9.4.2 Xây dựng Assembly.....	578
9.4.3 Xây dựng một ứng dụng Client .....	579
<b>9.5 Kết sinh một Proxy thông qua VS.NET .....</b>	<b>580</b>
<b>9.6 Một Web Service lý thú hơn cùng với Web Client .....</b>	<b>582</b>
9.6.1 Sản sinh hằng loạt kiểu dữ liệu custom .....	583
9.6.2 Cho biểu mẫu thêm phong phú .....	585
9.6.3 Xây dựng các kiểu dữ liệu serializable.....	587
<b>9.7 Tìm hiểu Discovery Service Protocol.....</b>	<b>589</b>
9.7.1 Thêm một Web Service mới .....	589
<b>CHỈ MỤC .....</b>	<b>591</b>





## *Lời mở đầu*

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

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

Bạn cứ thử hình dung là trong ngành điện toán, cứ vào khoảng một thập niên thì có một cuộc cách mạng nho nhỏ trong cách tiếp cận về lập trình. Vào thập niên 1960 là sự xuất hiện ngôn ngữ Cobol và Fortran (cũng như ngôn ngữ RPG của IBM) thay thế cho ngôn ngữ hợp ngữ; giữa thập niên 70 là sự xuất hiện máy vi tính với ngôn ngữ Basic; vào đầu thập niên 80 những công nghệ mới là Unix có thể chạy trên máy để bàn với ngôn ngữ cực mạnh mới là C, phát triển bởi ATT. Qua đầu thập niên 90 là sự xuất hiện của Windows và C++ (được gọi là C với

lớp), đi theo sau là khái niệm về lập trình thiên đối tượng trong bước khai mào. Mỗi bước tiến triển như thế tượng trưng cho một đợt sóng thay đổi cách lập trình của bạn: từ lập trình vô tổ chức qua lập trình theo cấu trúc (structure programming hoặc procedure programming), bây giờ qua lập trình thiên đối tượng. Lập trình thiên đối tượng trên C++ vẫn còn “khó nuốt” đối với những ai đã quen cái nếp nghĩ theo kiểu lập trình thiên cấu trúc. Và lại, lập trình thiên đối tượng vào cuối thập niên qua vẫn còn nhiều bất cập, không tự nhiên nên viết không thoải mái.

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

## ***Tập II được tổ chức thế nào?***

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

### **Chương 1: Xuất Nhập Dữ Liệu & Sản Sinh Hàng Loạt Đối Tượng**

.NET Framework cung cấp một số kiểu dữ liệu chuyên về các hoạt động Xuất/Nhập dữ liệu. Trong chương này bạn sẽ học cách cất trữ rồi tìm đọc lại những kiểu dữ liệu đơn giản từ các tập tin, ký ức và vùng đệm chuỗi. Một điểm quan trọng và khá lý thú là dịch vụ “sản sinh hàng loạt” (được gọi là serialization) đối tượng. Bằng cách sử dụng một tập hợp con các attribute

được định sẵn trước và một object graph tương ứng, .NET Framework có khả năng cho lưu tồn (persist) các đối tượng có liên hệ với nhau bằng cách sử dụng một XML hoặc binary formatter. Để minh họa việc sản sinh hàng loạt đối tượng hoạt động thế nào, chương này sẽ kết thúc bởi một ứng dụng Windows Forms (mang tên Car Logger) cho phép người sử dụng tạo và serialize các kiểu dữ liệu tự tạo dùng về sau.

## **Chương 2: Xây dựng một ứng dụng Windows**

C# cung cấp một mô hình RAD (Rapid Application Development) tương tự như trên ngôn ngữ Visual Basic. Chương này mô tả làm thế nào sử dụng một RAD để tạo những chương trình Windows chất lượng chuyên nghiệp sử dụng môi trường Windows Forms. Bạn sẽ hiểu làm thế nào xây dựng một cửa sổ chính stand-alone sử dụng các kiểu dữ liệu thuộc namespace System.Windows.Forms. Một khi bạn hiểu cách dẫn xuất một Form bạn sẽ học cách bổ sung việc hỗ trợ những hệ thống trình đơn tinh vi, thanh công cụ và thanh tình trạng. Ngoài ra, chương này cũng xét đến việc thao tác bằng lập trình system registry và Windows 2000 event log.

## **Chương 3: Tìm hiểu về Assembly và cơ chế Version**

Chương này minh họa việc làm thế nào chắt nhỏ một EXE nguyên khối thành những code library rời rạc. Bạn sẽ biết qua thành phần nội tại của một .NET assembly, phân biệt giữa “shared” và “private” assembly cũng như cách các assembly được tạo và quản lý thế nào. Trên .NET, một assembly là một tập hợp các tập tin mà người sử dụng xem như là một DLL hoặc EXE duy nhất. Assembly được xem như là đơn vị cơ bản trong việc tái sử dụng, đánh số phiên bản, bảo đảm an toàn sử dụng và triển khai hoạt động của một ứng dụng.

## **Chương 4: Tìm hiểu về Type Attributes và Reflection**

Các assembly .NET sử dụng rất nhiều metadata liên quan đến các lớp, các hàm hành sự, thuộc tính, và tình huống. Metadata này được biên dịch vào chương trình và có thể được tìm đọc lại thông qua phản chiếu (reflection). Chương này khảo sát việc làm thế nào thêm metadata vào đoạn mã, làm cách nào tạo những custom attribute, và làm thế nào truy cập metadata này thông qua phản chiếu. Trong chương này bạn sẽ biết qua triệu gọi động (dynamic

invocation) là gì, trong các hàm hành sự nào sẽ được triệu gọi với gắn kết trễ (late binding), cũng như việc minh họa bởi một reflection emit, một kỹ thuật cao cấp cho phép xây dựng những đoạn mã tự mình thay đổi biến hóa.

### **Chương 5: *Marshaling và Remoting***

.NET Framework được thiết kế chịu hỗ trợ các ứng dụng chạy trên Web cũng như phân tán cho nhiều người sử dụng. Các cấu kiện (component) được tạo ra trên C# có thể lưu trữ trên cùng máy hoặc trên các máy nằm xa xuyên qua mạng Internet. Chương này đề cập đến *Marshaling* một kỹ thuật cho phép tương tác với các đối tượng không thật sự nằm trên máy, và *Remoting* một kỹ thuật cho phép liên lạc với những đối tượng nằm ở xa này.

### **Chương 6: *Mạch Trình và Đồng Bộ Hóa***

Base Class Libraries đem lại một hỗ trợ khá phong phú đối với những xuất/nhập dữ liệu bất đồng bộ cũng như đối với các lớp khác làm cho việc thao tác tường minh các mạch trình không cần thiết. Tuy nhiên, C# cung cấp một hỗ trợ rất mạnh đối với mạch trình và đồng bộ hoá mà chương này sẽ đề cập đến.

### **Chương 7: *Tương tác với unmanaged code***

Với sự xuất hiện của .NET, giờ đây Component Object Model (COM) của Microsoft được xem như là công nghệ “di sản”, vì kiến trúc của COM không giống chút nào so với kiến trúc của .NET. Trong chương này, ta sẽ xét đến việc các kiểu dữ liệu COM và .NET “sống chung hoà bình” thế nào thông qua việc sử dụng COM Callable Wrapper (CCW) và Runtime Callable Wrapper (RCW). Trong chương này bạn sẽ thấy những cấu trúc IDL khác nhau (chẳng hạn SAFEARRAY, connection point và COM enumeration) được ánh xạ thế nào vào đoạn mã C#. Chương này sẽ kết thúc bằng cách xem xét cách làm thế nào xây dựng các kiểu dữ liệu .NET có thể tận dụng COM+ runtime.

### **Chương 8: *Lập trình ứng dụng Web với Web Forms và ASP .NET***

Chương 8 và chương 9 chỉ là những chương nhập môn cho tập V của bộ sách này. Chương này sẽ bắt đầu bởi một tổng quan về mô hình lập trình Web, xem xét việc tạo Web front end (HTML) thế nào, cách kiểm tra hợp lệ phía client cũng như yêu cầu trả lời từ một ứng dụng ASP cổ điển. Phần lớn chương này

dẫn dắt bạn vào kiến trúc của ASP.NET. Tại đây bạn sẽ học qua cách sử dụng các ô control Web cũng như việc thụ lý tình huống phía server.

## **Chương 9: *Lập trình Web Services***

Với chương cuối này, bạn sẽ biết qua vai trò của Web Services. Nói một cách đơn giản một “Web service” chỉ là một assembly được khởi động sử dụng một HTTP chuẩn. Tại đây bạn sẽ khám phá các công nghệ liên quan (WSDL, SOAP và Discovery Services) cho Web Service có khả năng nhận những yêu cầu từ phía khách hàng. Một khi bạn đã hiểu các tạo một C# Web Service, bạn có thể học cách xây dựng một lớp proxy phía client biết cất giấu phần logic chương trình SOAP cấp thấp khỏi view.

## ***Bộ sách gồm những tập nào?***

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

### **Tập I: *Lập trình Visual C# thế nào?***

Chương 1	Visual C# và .NET Framework
Chương 2	Bắt đầu từ đây ta tiến lên!
Chương 3	Sử dụng Debugger thế nào?
Chương 4	Căn bản Ngôn ngữ C#
Chương 5	Lớp và Đối tượng
Chương 6	Kế thừa và Đa hình
Chương 7	Nạp chồng tác tử
Chương 8	Cấu trúc Struct
Chương 9	Giao diện
Chương 10	Bản dãy, Indexers và Collections
Chương 11	Chuỗi chữ và biểu thức regular
Chương 12	Thụ lý các biệt lệ
Chương 13	Ủy thác và tình huống
Chương 14	Lập trình trên môi trường .NET

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

Chương 1	Tạo giao diện người sử dụng dùng lại được
Chương 2	Thiết kế giao diện theo Lớp và Tam nguyên

- Chương 3 Tìm hiểu đồ hoạ và GDI+
- Chương 4 Tìm hiểu biểu mẫu
- Chương 5 Cơ bản về lớp Control
- Chương 6 Windows Forms Controls
- Chương 7 Các ô control tiên tiến
- Chương 8 Custom Controls
- Chương 9 Hỗ trợ Custom Control vào lúc thiết kế
- Chương 10 MDI Interfaces và Workspace
- Chương 11 Dynamic User Interfaces
- Chương 12 Data Controls
- Chương 13 GDI+ Controls
- Chương 14 Hỗ trợ Help

## **Tập IV: Lập trình Căn cứ Dữ liệu với Visual C# & ADO.NET**

- Chương 1 Sử dụng Căn cứ dữ liệu
- Chương 2 Tổng quan về ADO .NET
- Chương 3 Data Component trong Visual Studio .NET
- Chương 4 Các lớp ADO.NET tách rời
- Chương 5 ADO.NET Data Providers
- Chương 6 Trình bày IDE theo quan điểm Database
- Chương 7 Làm việc với XML
- Chương 8 Triển khai ứng dụng Web sử dụng ADO.NET
- Chương 9 Sử dụng các dịch vụ Web với ADO.NET
- Chương 10 Thụ lý các tình huống trên ADO.NET
- Chương 11 Stored procedure, View, Trigger & Com Interop
- Chương 12 Làm việc với Active Directory
- Chương 13 Tìm hiểu về Message Queues
- Chương 14 Tìm hiểu về Data Wrapper
- Chương 15 Làm việc với ODBC.NET data provider

## **Tập V: Lập trình ASP.NET & Web**

- Chương 1 ASP.NET và NET Framework
- Chương 2 Tìm hiểu các tình huống
- Chương 3 Tìm hiểu các ô Web Control
- Chương 4 Chi tiết về các ASP Control
- Chương 5 Lập trình Web Form

- Chương 6 Kiểm tra hợp lệ
- Chương 7 Gắn kết dữ liệu
- Chương 8 Tìm hiểu về DataGrid Control
- Chương 9 Truy cập căn cứ dữ liệu với ADO.NET
- Chương 10 ADO.NET Data Update
- Chương 11 Tìm hiểu về Repeatre & DataList Control
- Chương 12 User Control và Custom Control
- Chương 13 Web Services
- Chương 14 Caching và Năng suất
- Chương 15 An toàn
- Chương 16 Triển khai ứng dụng

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

- Chương 01 Tổng quan về Crystal Reports .Net
- Chương 02 Hãy thử bắt đầu với Crystal Reports .NET
- Chương 03 Tìm hiểu Crystal Reports Object Model
- Chương 04 Sắp xếp & Gộp nhóm
- Chương 05 Sử dụng các thông số
- Chương 06 Uốn nắn các báo cáo
- Chương 07 Tìm hiểu về Công thức & Lô gic chương trình
- Chương 08 Vẽ biểu đồ thế nào?
- Chương 09 Tạo báo cáo Cross-Tab
- Chương 10 Thêm Subreports vào báo cáo chính
- Chương 11 Hội nhập báo cáo vào ứng dụng Windows
- Chương 12 Hội nhập báo cáo vào ứng dụng Web
- Chương 13 Tạo XML Report Web Services
- Chương 14 Làm việc với các dữ liệu nguồn
- Chương 15 Xuất khẩu và triển khai hoạt động các báo cáo

## **Tập VII:       Sổ tay kỹ thuật C# - phần A**

Chưa định hình các chương

## **Tập VIII:      Sổ tay kỹ thuật C# - phần B**

Chưa định hình các chương

## ***Bộ sách này dành cho ai?***

Bộ sách này được viết dành cho những ai muốn triển khai những ứng dụng chạy trên Windows hoặc trên Web dựa trên nền .NET. Chắc chắn là có nhiều bạn đã quen viết C++, Java hoặc Visual Basic, hoặc Pascal. Cũng có thể bạn đọc khác lại quen với một ngôn ngữ khác hoặc chưa có kinh nghiệm gì về lập trình ngoài lập trình cơ bản. Bộ sách này dành cho tất cả mọi người. Vì đây là một bộ sách tự học không cần thầy, chỉ cần có một máy tính được cài đặt .NET. Đối với ai chưa hề có kinh nghiệm lập trình thì hơi khó một chút nhưng “cày đi cày lại” thì cũng vượt qua nhanh những khó khăn này. Còn đối với những ai đã có kinh nghiệm lập trình thì sẽ mê ngay ngôn ngữ này và chỉ trong một thời gian rất ngắn, 6 tháng là tối đa là có thể nắm vững những góc ngách của ngôn ngữ mới này, và có thể biết đầu chùng trong một thời gian rất ngắn bạn sẽ trở thành một “guru” ngôn ngữ C#.

Người viết cũng xin lưu ý bạn đọc là bộ sách này là sách tự học (tutorial) chứ không phải một bộ sách tham chiếu (reference) về ngôn ngữ, nên chỉ mở đường phát quang hướng dẫn bạn đi khỏi bị lạc, và đem lại 60% kiến thức về ngôn ngữ. Và khi học tới đâu, tới một chặng đường nào đó bạn có thể lên MSDN phẳng lần đào sâu từng đề mục con mà bạn đang còn mơ hồ để có thể phẳng lần 40% còn lại kiến thức để nắm vững vấn đề.

Lấy một thí dụ. Trong bộ sách này, chúng tôi thường đề cập đến các lớp. Chúng tôi giải thích tổng quát cho biết lớp sẽ được dùng vào việc gì và sử dụng một số hàm hành sự (method) hoặc thuộc tính (property) tiêu biểu của lớp này trong những thí dụ cụ thể. Thế nhưng mỗi lớp có vô số hàm hành sự và thuộc tính cũng như tình huống. Thì lúc này bạn nên vào MSDN tham khảo từng hàm hành sự hoặc thuộc tính một của lớp này để bạn có một ý niệm sơ sơ về những công năng và đặc tính của lớp. Có một số chức năng/đặc tính bạn sẽ chẳng bao giờ sử dụng đến, còn một số thì thoảng bạn mới cần đến.

Cho nên về sau, khi bạn muốn thực hiện một chức năng gì đó, thì bạn có thể vào lại MSDN xem lớp có một hàm hoặc thuộc tính đáp ứng đúng (hoặc gần đúng) nhu cầu của bạn hay không và nếu có thì lúc này bạn mới xem kỹ cách sử dụng. Kinh nghiệm cho thấy, là trong suốt cuộc đời hành nghề lập trình viên, bạn sẽ “xào đi xào lại” cũng chừng này lệnh, hoặc một số hàm nào đó theo một mẫu



dáng (pattern) nào đó, nên một khi bạn đã khám phá ra những lệnh hoặc hàm này, và áp dụng thành công thì bạn sẽ thường xuyên dùng đến một cách máy móc không cần suy nghĩ gì thêm.

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

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

Ngoài ra, trong tương lai, nếu sức khỏe cho phép (vì dù gì thì tuổi người viết cũng gần 72) chúng tôi dự kiến ra bộ sách về phân tích thiết kế các ứng dụng điện toán sử dụng UML và Pattern. Trong những buổi gặp gỡ giữa bạn bè và một đôi lần trên báo chí khi họ than phiền là kỹ sư tin học bây giờ ra trường không sử dụng được, chúng tôi thường hay phát biểu là không ngạc nhiên cho lắm khi ta chỉ cho ra toàn là “thợ lập trình” giống như bên xây dựng là thợ hồ, thợ nề thợ điện thợ mộc v.v.. chứ đâu có đào tạo những kiến trúc sư (architect) biết thiết kế những bản vẽ hệ thống. Do đó, chúng tôi dự kiến (hy vọng là như vậy) là sẽ hoàn thành một bộ sách đề cập đến vấn đề phân tích thiết kế những hệ thống sử dụng những thành tựu mới nhất trên thế giới là UML và Pattern với những phần mềm thích ứng là IBM Rational Rose XDE và Microsoft Visio for Enterprise Architect.

Ngoài ra, những gì học ở trường là thuần túy về kỹ thuật lập trình, về mô phỏng, trí tuệ nhân tạo, lý thuyết rời rạc v.v.. (mà những mớ lý thuyết này không có đất dụng võ) nhưng khi ra trường vào các xí nghiệp thì mù tịt về quản lý nhân sự, về kế toán về tồn kho vật tư, về tiêu thụ v.v.. mà 80% ứng dụng tin học lại là vào các lãnh vực này. Do đó, trong bộ sách mà chúng tôi dự kiến sẽ soạn những

tập đi sâu vào xây dựng những hệ thống quản lý trong các cơ quan xí nghiệp hành chính cũng như thương mại.

## ***Đôi lời cuối cùng***

Kể từ năm 1989, năm thành lập văn phòng dịch vụ điện toán SAMIS, cho đến nay gần trên 15 năm chúng tôi cùng anh chị em trong nhóm SAMIS đã biên soạn trên 55 đầu sách, và cũng đã phát hành gần 400.000 bản, trong ấy 60% là phần của người viết. Từ những tiền lời kiếm được do tự phát hành lấy cộng thêm tiền hưu tiết kiệm của bà vợ người Thụy Sĩ, hằng năm chúng tôi đã dành toàn bộ số tiền này để xây các trường cho những vùng sâu vùng xa trên 15 tỉnh thành đất nước (Sơn La, Nghệ An, Quảng Ngãi, Quảng Nam, Quảng Trị, Bình Định, Ban Mê Thuột, Pleiku, Daklak, Bà Rịa Vũng Tàu, Đồng Nai, Bình Dương, TP. Hồ Chí Minh, Cần Thơ, và Cà Mau), cấp học bổng cho các sinh viên nghèo tại các đại học Huế, Đà Nẵng, An Giang và TP. Hồ Chí Minh, hỗ trợ vốn cho giáo viên ở An Lão (Bình Định), xây nhà cho giáo viên ở vùng sâu vùng xa (Bình Định, Quảng Trị), và tài trợ mổ mắt cho người nghèo ở An Giang (4 năm liền).

Các hoạt động xã hội này đều thông qua sự hỗ trợ của hai tờ báo Tuổi Trẻ và Sài Gòn Giải Phóng. Không ngờ những việc làm rất cá nhân này lại được Nhà Nước “theo dõi” đến nỗi không biết vị nào đã “xúi” Chủ tịch nước Trần Đức Lương ký quyết định tặng người viết Huân chương Lao động Hạng 3, ngày 29/8/2000. Nói ra điều này, chúng tôi muốn bạn đọc hiểu cho là tự nội lực của ta, ta cũng có thể giúp đỡ giáo dục mà khỏi nhờ viện trợ của các nước Nhật, Hàn Quốc. Nếu các bạn ý thức rằng mỗi tập sách bạn mua của chúng tôi thay vì mua sách “luộc” hoặc photocopy là bạn đã gián tiếp tham gia vào chương trình xây trường lớp cho vùng sâu vùng xa cũng như hỗ trợ học bổng cho sinh viên nghèo của chúng tôi.

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

TP. Hồ Chí Minh ngày 1/3/2005

**Dương Quang Thiện**

# Chương 1

## Xuất Nhập Dữ Liệu & Sản Sinh Hàng Loạt Đối Tượng

Đối với nhiều ứng dụng, dữ liệu thường được trữ trong ký ức và bạn có thể truy xuất nó như là một đồ vật 3 chiều; khi bạn cần một biến hoặc một đối tượng bạn chỉ cần gọi tên là nó có mặt liền lập tức. Tuy nhiên, khi bạn muốn di chuyển dữ liệu của mình vào ra một tập tin, hoặc xuyên qua mạng Internet chẳng hạn, thì dữ liệu này phải ở dưới dạng một *dòng chảy* (stream) theo thành từng *bè* (packet), nghĩa là trên một *stream* những *bè dữ liệu* (packet of data flow) xuôi theo dòng chảy giống như những bè gỗ xuôi dòng sông.

Điểm cuối xuất phát của một dòng chảy là một “kho trữ hậu phương” (backing store), cung cấp một nguồn dữ liệu đối với dòng chảy, giống như hồ chứa là nguồn nước của một dòng sông. Điển hình, backing store là một tập tin, nhưng cũng có thể là một mạng lưới hoặc một kết nối với Web.

Các tập tin (file) và thư mục (directory) được trừu tượng hóa bởi những lớp trong .NET Framework. Các lớp này bao gồm những hàm hành sự và thuộc tính cho phép bạn tạo, đặt tên, thao tác và xóa sổ các tập tin và thư mục trên đĩa của bạn.

.NET Framework cung cấp cả những dòng chảy có ký ức đệm (buffered stream) cũng như dòng chảy không có ký ức đệm (unbuffered stream), cũng như cung cấp những lớp lo việc xuất nhập dữ liệu bất đồng bộ (asynchronous I/O). Với xuất nhập dữ liệu bất đồng bộ bạn có thể ra lệnh cho các lớp .NET đọc tập tin, và khi chúng đang bạn đọc dữ liệu từ đĩa thì chương trình của bạn có thể quay qua làm việc gì khác, và xuất nhập dữ liệu bất đồng bộ sẽ thông báo cho bạn biết khi chúng xong việc. Vì các lớp bất đồng bộ đủ mạnh và “chắc nịch”, bạn có thể tránh việc tạo rõ ra những mạch trình (thread).

Việc xuất nhập dữ liệu khỏi các tập tin cũng không khác gì khi xuôi dòng chảy trên mạng và phần hai của chương này sẽ mô tả streaming sử dụng nghi thức TCP/IP và Web.

Muốn tạo một dòng chảy dữ liệu, đối tượng của bạn phải được *sản sinh hàng loạt* (serialized), hoặc viết thành dòng chảy gồm một loạt bit. .NET Framework cung cấp một hỗ trợ khá phong phú liên quan đến việc sản sinh hàng loạt, và phần cuối của chương này sẽ dẫn dắt bạn qua những chi tiết giúp bạn nắm vững việc sản sinh hàng loạt các đối tượng của bạn.

## 1.1. Các tập tin và Thư mục

Trước khi xem làm thế nào bạn có thể lấy dữ liệu từ các tập tin hoặc viết dữ liệu lên tập tin, ta bắt đầu xét đến việc .NET hỗ trợ thế nào trong những thao tác liên quan các tập tin và thư mục. Trên .NET Framework, namespace **System.IO** là vùng của các thư viện lớp dành cho những dịch vụ liên quan đến xuất nhập dữ liệu dựa trên tập tin (và dựa trên ký ức). Giống như bất cứ namespace nào, **System.IO** định nghĩa một số lớp, enumeration, structure, và delegate, tất cả nằm trong **mscorlib.dll**.

### 1.1.1 Khảo sát namespace System.IO

Như bạn sẽ thấy trong chương này, các lớp trong namespace **System.IO** tập trung phần lớn vào việc thao tác các tập tin và thư mục (directory; Windows còn gọi là folder). Tuy nhiên, trong namespace này cũng có những lớp hỗ trợ việc đọc viết dữ liệu lên những vùng chuỗi chữ kể cả những vị trí ký ức thô. Để bạn có cái nhìn toàn diện về chức năng của namespace **System.IO**, bảng 1-1 phác thảo những lớp (nonabstract) cốt lõi.

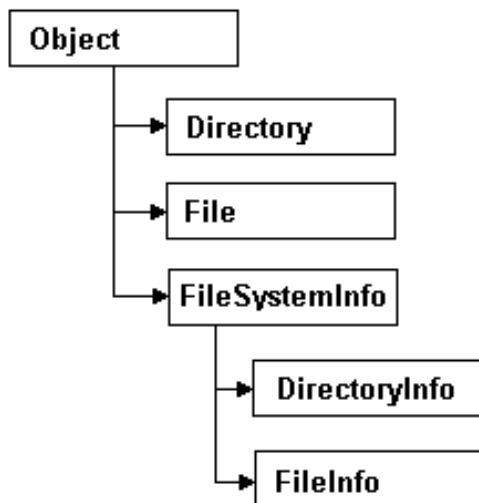
**Bảng 1-1: Các lớp cốt lõi của namespace System.IO**

Các lớp	Ý nghĩa
<b>BinaryReader</b> <b>BinaryWriter</b>	Hai lớp này cho phép bạn trữ (BinaryWriter) hoặc tìm đọc lại (BinaryReader) những kiểu dữ liệu nguyên sinh (integer, boolean, string, v.v..) như là những trị nhị phân.
<b>BufferedStream</b>	Lớp này cung cấp một chỗ trữ tạm thời cho một dòng dữ liệu (stream) kiểu bytes, và có thể biến thành chỗ trữ về sau.
<b>Directory</b> <b>DirectoryInfo</b> <b>File</b> <b>FileInfo</b>	Các lớp này dùng thao tác những thuộc tính liên quan đến một thư mục nào đó hoặc tập tin vật lý kể cả việc tạo những tập tin mới và nói rộng cấu trúc thư mục hiện hành. Các lớp <b>Directory</b> và <b>File</b> trưng ra những chức năng như là những hàm hành sự static (nghĩa là bạn khởi tạo một thể hiện đối tượng). Còn <b>DirectoryInfo</b> và <b>FileInfo</b> cũng trưng ra chức năng tương tự từ một thể hiện đối tượng hợp lệ.
<b>FileStream</b>	Lớp này cho phép truy xuất trực truy (random file access), với dữ liệu được trình bày như là một dòng dữ liệu kiểu byte. Với lớp này bạn có những khả năng truy tìm (seek capabilities).
<b>MemoryStream</b>	Lớp này cho phép truy xuất trực truy dòng dữ liệu được trữ trên ký ức thay vì trên một tập tin vật lý.

<b>NetworkStream</b>	Một dòng dữ liệu trên một kết nối mạng.
<b>Stream</b>	Lớp abstract chịu hỗ trợ việc đọc/viết những bytes dữ liệu.
<b>StreamWriter</b> <b>StreamReader</b>	Hai lớp này dùng để trữ ( <b>StreamWriter</b> ) hoặc tìm đọc lại ( <b>StreamReader</b> ) những thông tin dạng văn bản trữ trên một tập tin. Các lớp này không cho phép truy xuất tập tin theo trực truy.
<b>StringWriter</b> <b>StringReader</b>	Giống như hai lớp kể trên <b>StreamWriter/ StreamReader</b> , hai lớp này cũng làm việc với thông tin dạng văn bản được trữ trên ký ức, thay vì trên tập tin vật lý.
<b>TextReader</b> <b>TextWriter</b> <b>StringReader</b> <b>StringWriter</b>	<b>TextReader</b> và <b>TextWriter</b> là những lớp abstract được thiết kế dành cho ký tự I/O Unicode. Còn <b>StringReader</b> và <b>StringWriter</b> cho phép xuất nhập liệu dưới dạng stream hoặc string.

Ngoài những lớp kể trên, còn có những enumeration và lớp abstract (Stream, TextReader, TextWriter, v.v..) định nghĩa một giao diện đa hình chia sẻ sử dụng cho tất cả các lớp hậu duệ. Trong chương này bạn sẽ làm quen với những lớp này.

## 1.1.2 Các lớp Directory(Info) và File(Info)



**Hình 1-01: Các lớp File và Directory**

Theo sơ đồ hình 1-1, bạn thấy là **Directory** và **File** được dẫn xuất từ lớp **System.Object**,

**System.IO** cung cấp cho bạn 4 lớp (Directory, File, DirectoryInfo và FileInfo) cho phép bạn thao tác với các tập tin riêng rẽ cũng như tương tác với cấu trúc thư mục của máy. Hai lớp đầu tiên **Directory** và **File** bao gồm những thành viên static cho phép những tác vụ tạo, gỡ bỏ và thao tác khác nhau trên các tập tin và trên thư mục. Vì là những thành viên static nên bạn có thể triệu gọi các thành viên này khỏi phải thể hiện đối tượng lớp. Hai lớp có liên hệ mật thiết **DirectoryInfo** và **FileInfo** thì cũng trưng ra những chức năng tương tự như với hai lớp **Directory** và **File**, nhưng vì các thành viên không phải là static, nên muốn triệu gọi các thành viên bạn phải tạo trước tiên một thể hiện đối tượng lớp.

trong khi **DirectoryInfo** và **FileInfo** thì lại dẫn xuất từ **FileSystemInfo**. Lớp **FileSystemInfo**, một lớp cơ bản abstract, có một số thuộc tính và hàm hành sự cung cấp những thông tin liên quan đến một tập tin hoặc thư mục.

### 1.1.2.1 Lớp cơ bản abstract *FileSystemInfo*

Hai lớp **DirectoryInfo** và **FileInfo** thừa hưởng một số hành vi từ lớp cơ bản abstract **FileSystemInfo**, thường là để biết những đặc tính chung (chẳng hạn thời gian thành lập, những thuộc tính khác nhau v.v..) liên quan đến một tập tin hoặc thư mục. Bảng 1-2 liệt kê một số thuộc tính cốt lõi của **FileSystemInfo**.

**Bảng 1-2: Các thuộc tính cốt lõi của lớp cơ bản abstract *FileSystemInfo* thuộc tính**

Các thuộc tính	Ý nghĩa
<b>Attributes</b>	Đi lấy hoặc đặt để những attribute được gắn liền với tập tin hiện hành. Trị Attributes lấy từ enumeration <b>FileAttributes</b> (xem bảng 1-4).
<b>CreationTime</b>	Đi lấy hoặc đặt để thời gian tạo tập tin hoặc thư mục hiện hành.
<b>Exists</b>	Thuộc tính này có thể dùng để xác định liệu xem một tập tin hoặc thư mục hiện hữu hay không.
<b>Extension</b>	Dùng để tìm lại phần đuôi của một tập tin.
<b>FullName</b>	Đi lấy lối tìm về trọn vẹn của một tập tin hoặc thư mục.
<b>LastAccessTime</b>	Đi lấy hoặc đặt để thời gian mà tập tin hoặc thư mục hiện hành được truy xuất lần chót.
<b>LastWriteTime</b>	Đi lấy hoặc đặt để thời gian khi tập tin hoặc thư mục hiện hành được viết lên lần chót.
<b>Name</b>	Thuộc tính này trả về tên của một tập tin; là một thuộc tính read-only. Đối với một thư mục thuộc tính này đi lấy tên của thư mục chót trên đẳng cấp nếu có thể được; bằng không thì đi tìm lại tên chính danh trọn vẹn (fully qualified name).

Lớp **FileSystemInfo** cũng định nghĩa một hàm hành sự **Delete()**, được thi công bởi những lớp dẫn xuất để gỡ bỏ một tập tin hoặc thư mục nào đó khỏi ổ đĩa. Ngoài ra, còn

có một hàm hành sự **Refresh()** mà bạn có thể triệu gọi trước khi đi lấy thông tin thuộc tính bảo đảm là thông tin không lỗi thời.

### 1.1.2.2 Làm việc với lớp *DirectoryInfo*

Lớp đầu tiên bạn cần tìm hiểu là **DirectoryInfo**. Lớp này có một số thành viên dùng để tạo, di chuyển, gỡ bỏ và rảo xem các thư mục và thư mục con. Ngoài những chức năng do lớp **FileSystemInfo** cung cấp, **DirectoryInfo** còn có một số thành viên được liệt kê bởi bảng 1-3.

**Bảng 1-3: Các thành viên của lớp DirectoryInfo**

Các thành viên	Ý nghĩa
<b>Create()</b> <b>CreateSubdirectory()</b>	Các hàm này tạo một thư mục (hoặc thư mục con) theo tên một lối tìm về (path).
<b>Delete()</b>	Gỡ bỏ một thư mục và luôn tất cả nội dung của thư mục.
<b>GetDirectories()</b>	Hàm này trả về một bản dãy chuỗi tượng trưng cho tất cả các thư mục con trên thư mục hiện hành.
<b>GetFiles()</b>	Hàm này đi lấy các tập tin nằm trong thư mục được khai báo (như là một bản dãy kiểu <b>FileInfo</b> ).
<b>MoveTo()</b>	Di chuyển một thư mục và nội dung thư mục về một path mới.
<b>Parent</b>	Thuộc tính này cho biết thư mục cha-mẹ của lối tìm về được khai báo.

### 1.1.2.3 Tạo một đối tượng *DirectoryInfo*

Muốn khảo sát một cây đẳng cấp thư mục (directory hierarchy), bạn cần hiển lộ (instantiate) một đối tượng **DirectoryInfo**. Lớp **DirectoryInfo** không những cung cấp các hàm hành sự đi lấy tên các tập tin và thư mục mà còn những đối tượng **FileInfo** và **DirectoryInfo**, cho phép bạn lũng sục sâu vào cấu trúc đẳng cấp các thư mục để lôi ra những thư mục con cũng như khảo sát các thư mục con này một cách đệ quy.

Bạn bắt đầu làm việc với lớp **DirectoryInfo** bằng cách khai báo một lối tìm về thư mục cụ thể (chẳng hạn “C:\”, “D:\WINNT”, \\CompanyServer\\Utils, “A:\”, v.v..), như là thông số của hàm constructor. Nếu bạn muốn truy cập thư mục hiện dịch (active

directory, nghĩa là thư mục của ứng dụng đang thi hành), bạn dùng ký hiệu “.”. Sau đây là một vài thí dụ:

```
// Tạo một thư mục mới từ thư mục hiện hành trở đi
DirectoryInfo dir1 = new DirectoryInfo(".");

// Tạo một thư mục mới từ C:\Foo\Bar trở đi
DirectoryInfo dir2 = new DirectoryInfo(@"C:\Foo\Bar");
```

**Bạn để ý** Dấu @ sử dụng trên thí dụ thứ hai tạo một chuỗi verbatim literal theo đây bạn khỏi dùng đến các ký tự escape như backslash “\”.

Nếu bạn cố ánh xạ một thư mục không hiện hữu, thì các lệnh trên sẽ tung ra một biệt lệ kiểu **System.IO.DirectoryNotFoundException**. bạn giả định là không có biệt lệ này được tung ra, bạn có thể khảo sát nội dung nằm sau thư mục sử dụng bất cứ thuộc tính nào được kế thừa từ **FileInfo**. Để minh họa, lớp sau đây tạo một kiểu **DirectoryInfo** mới ánh xạ lên “C:\WINNT” rồi cho tuôn in ra những thông kê đáng chú ý. Thí dụ 1-1 cho thấy bảng liệt in và hình 1-1 là kết xuất:

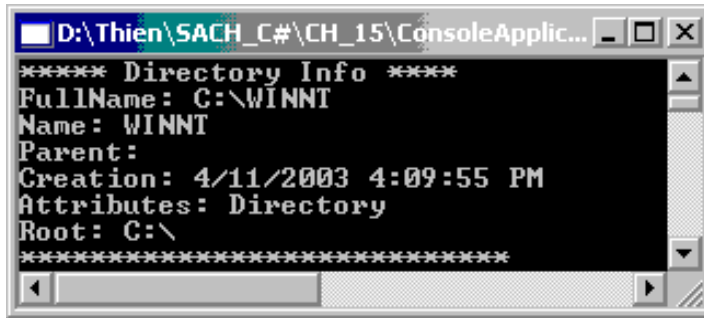
### ***Thí dụ 1-1: In ra thông tin liên quan đến thư mục C:\WINNT***

```
*****
namespace Prog_CSharp
{
    using System;
    using System.IO;

    class MyDirectory
    {
        static void Main(string[] args)
        {
            DirectoryInfo dir = new DirectoryInfo(@"C:\WINNT");
            Console.WriteLine("***** Directory Info *****");
            Console.WriteLine("FullName: {0}", dir.FullName);
            Console.WriteLine("Name: {0}", dir.Name);
            Console.WriteLine("Parent: {0}", dir.Parent);
            Console.WriteLine("Creation: {0}", dir.CreationTime);
            Console.WriteLine("Attributes: {0}", dir.Attributes.ToString());
            Console.WriteLine("Root: {0}", dir.Root);
            Console.WriteLine("*****");
        }
    }
}
*****
```

Hình 1-2 cho thấy kết xuất:





Hình 1-2: Thông tin thư mục C:\WINNT

### 1.1.2.4 Enumeration FileAttributes

Trong thí dụ 1-1 trên, thuộc tính **Attribute** cho biết thuộc tính của thư mục hoặc của tập tin. Trị này lấy từ enum **FileAttributes**. Bảng 1-4 cho liệt kê các trị của enum này.

Bảng 1-4: Các trị của Enumeration FileAttributes

Member name	Description
<b>Archive</b>	Cho biết tình trạng lưu trữ, Archive, của tập tin. Các ứng dụng dùng thuộc tính này để đánh dấu các tập tin dành cho sao phòng hồ (backup) hoặc gỡ bỏ.
<b>Compressed</b>	Cho biết tập tin bị nén.
<b>Device</b>	Dành cho sử dụng về sau.
<b>Directory</b>	Tập tin là một thư mục.
<b>Encrypted</b>	Tập tin hoặc thư mục bị mật mã hóa. Đối với một tập tin, có nghĩa là tất cả dữ liệu trong tập tin bị mật mã hoá. Đối với một thư mục, đây có nghĩa mật mã hóa là trị mặc nhiên đối với những tập tin và thư mục được tạo mới.
<b>Hidden</b>	Tập tin bị cất giấu, và như vậy không bao gồm trong bảng liệt kê thư mục thông thường.
<b>Normal</b>	Tập tin này là bình thường, không thuộc tính nào được đặt để. Chỉ hợp lệ nếu sử dụng một mình.
<b>NotContentIndexed</b>	Tập tin sẽ không bị chỉ mục hoá bởi dịch vụ chỉ mục nội dung của hệ điều hành.
<b>Offline</b>	Tập tin này là offline. Dữ liệu của tập tin không có sẵn ngay liền.
<b>ReadOnly</b>	Tập tin này chỉ đọc mà thôi.
<b>ReparsePoint</b>	Tập tin này chứa một reparse point, là một khối dữ liệu tự tạo (user-defined) được gắn liền với một tập tin hoặc một thư mục.

<b>SparseFile</b>	Tập tin là một sparse file. Sparse file là những tập tin đồ sộ mà dữ liệu gồm phần lớn là zeros.
<b>System</b>	Tập tin này là một tập tin hệ thống. Tập tin là thành phần của hệ điều hành hoặc dành cho hệ điều hành độc quyền sử dụng.
<b>Temporary</b>	Tập tin này là tạm thời. File systems cố gắng giữ tất cả các dữ liệu trong ký ức để có thể truy xuất nhanh thay vì tuần ghi lên đĩa. Một tập tin tạm thời phải được gỡ bỏ bởi ứng dụng khi không dùng đến nữa.

## 1.1.3 Rảo xem các tập tin thông qua lớp DirectoryInfo

Bạn có thể nói rộng lớp MyDirectory trên thí dụ 1-1, bằng cách dùng một số hàm hành sự của lớp **DirectoryInfo**. Trước tiên, bạn sử dụng đến hàm hành sự **GetFiles()** để đọc tất cả các tập tin \*.bmp trên thư mục C:\WINNT chẳng hạn. Hàm này sẽ trả về một bản dãy kiểu **FileInfo**, và bạn có thể rảo xem bản dãy này thông qua lệnh foreach, như theo thí dụ 1-2 sau đây, và hình 1-3 kết xuất khi cho chạy chương trình này:

### *Thí dụ 1-2: In ra các tập tin \*.bmp thuộc thư mục*

```
*****
using System;
using System.IO;
namespace ConsoleApplication1
{
    class MyDirectory
    {
        static void Main(string[] args)
        {
            DirectoryInfo dir = new DirectoryInfo(@"C:\WINNT\");
            Console.WriteLine("***** Directory Info *****");
            Console.WriteLine("FullName: {0}", dir.FullName);
            Console.WriteLine("Name: {0}", dir.Name);
            Console.WriteLine("Parent: {0}", dir.Parent);
            Console.WriteLine("Creation: {0}", dir.CreationTime);
            Console.WriteLine("Attributes: {0}",
                dir.Attributes.ToString());
            Console.WriteLine("Root: {0}", dir.Root);
            Console.WriteLine("*****\n");

            // Đi lấy tất cả các tập tin mang đuôi *.bmp
            FileInfo[] bmpFiles = dir.GetFiles("*.bmp");

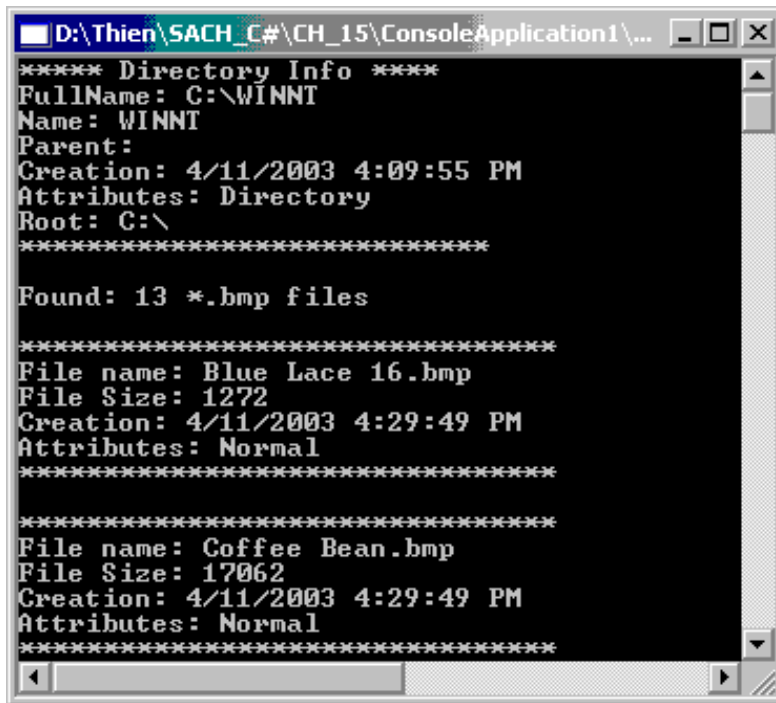
            // Tìm được bao nhiêu?
            Console.WriteLine("Found: {0} *.bmp files\n",
                bmpFiles.Length);

            // Bây giờ in ra tuần tự mỗi tập tin một
            foreach (FileInfo f in bmpFiles)
```

```

    {
        Console.WriteLine("*****");
        Console.WriteLine("File name: {0}", f.Name);
        Console.WriteLine("File Size: {0}", f.Length);
        Console.WriteLine("Creation: {0}", f.CreationTime);
        Console.WriteLine("Attributes: {0}",
            f.Attributes.ToString());
        Console.WriteLine("*****\n");
    }
    Console.ReadLine(); // để cho màn hình nằm yên
}
}
}
*****

```



Hình 1-3 : Thông tin liên quan đến các tập tin bitmap

## 1.1.4 Rảo xem các thư mục con

Như bạn có thể thấy trên thí dụ 1-1, bạn có thể hỏi thông tin bản thân đối tượng **DirectoryInfo** bao gồm tên của nó, lối tìm về, thuộc tính, thời gian truy xuất lần chót v.v.. Muốn khảo sát cây đẳng cấp thư mục con (subdirectory hierarchy), bạn hỏi thư mục hiện hành cho biết danh sách các thư mục con:

// Danh sách các thư mục con thuộc thư mục hiện hành

DirectoryInfo[] directories = dir.GetDirectories();

Lệnh trên trả về một bản đầy các đối tượng **DirectoryInfo** mỗi đối tượng tượng trưng cho một thư mục. Sau đó, bạn có thể đệ quy vào cùng hàm hành sự, trao qua mỗi lần một đối tượng **DirectoryInfo**:

```
foreach (DirectoryInfo newDir in directories)
{
    dirCounter++;
    ExploreDirectory(newDir);
}
```

Biến thành viên số nguyên static **dirCounter** dùng theo dõi bao nhiêu thư mục con được tìm thấy. Để thêm hấp dẫn bạn dùng thêm một biến số nguyên static thứ hai, **indentLevel**, dùng theo dõi cấp bậc thư mục. Bạn tăng 1 đối với trị của indentLevel mỗi lần bạn chui vào đệ quy một subdirectory, và giảm đi khi bạn thoát khỏi. Việc này cho phép bạn hiển thị những thư mục con canh thụt (indent) dưới thư mục cha-mẹ. Bảng liệt in 1-3 cho thấy toàn bộ chương trình rảo xem các thư mục con, và hình 1-4 là kết xuất:

### ***Thí dụ 1-3: Đệ quy xuyên các thư mục con***

\*\*\*\*\*

```
using System;
using System.IO;
namespace Prog_CSharp
{
    class MyDirectory
    {
        static void Main(string[] args)
        {
            MyDirectory t = new MyDirectory();
            // chọn thư mục con ban đầu
            string theDirectory = @"D:\THIEN\SACH_C#";
            // triệu gọi hàm khảo sát thư mục, hiển thị ngày
            // truy xuất và tất cả các thư mục con
            DirectoryInfo dir = new DirectoryInfo(theDirectory);
            t.ExploreDirectory(dir);
            // Xong! In ra tổng số thư mục tìm thấy được
            Console.WriteLine("\n\n{0} directories found.\n",
                               dirCounter);
        }
        // Hàm đệ quy đối với mỗi thư mục
        private void ExploreDirectory(DirectoryInfo dir)
        {
            indentLevel++; // ấn xuống một cấp directory

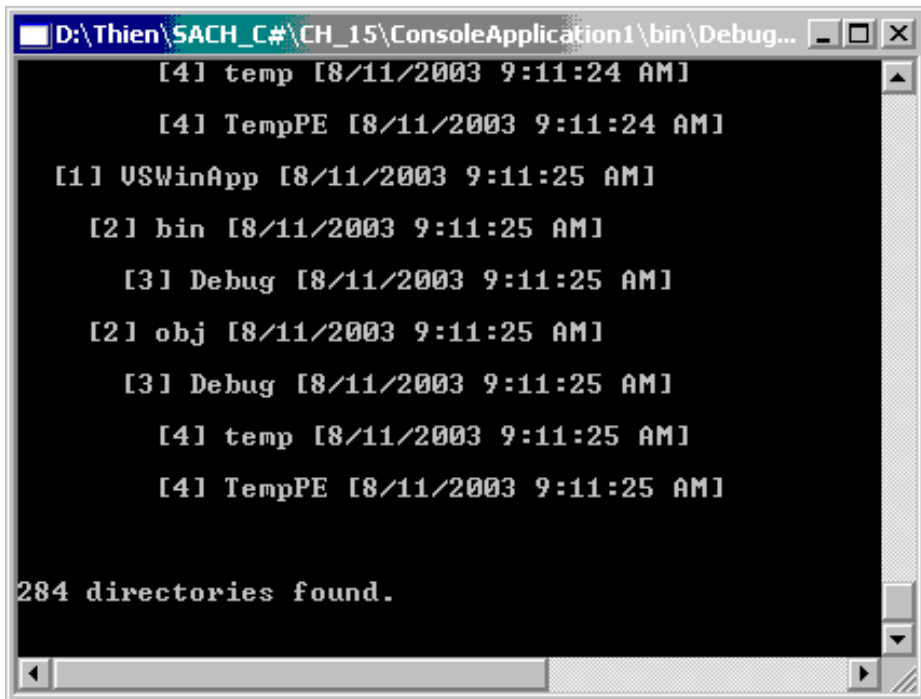
            // tạo indentation cho thư mục con
            for (int i = 0; i < indentLevel; i++)
            {
                Console.Write(" "); // hai space cho mỗi level
            }

            // in ra thư mục và thời gian truy xuất chót nhất
            Console.WriteLine("[{0}] {1} [{2}]\n",
                               indentLevel, dir.Name, dir.LastAccessTime);
        }
    }
}
```

```

        // đi lấy tất cả các thư mục trên thư mục hiện hành
        // rồi triệu gọi hàm này một cách đệ quy
        DirectoryInfo[] directories = dir.GetDirectories();
        foreach(DirectoryInfo newDir in directories)
        {
            dirCounter++;
            ExploreDirectory(newDir);
        }
        indentLevel--;
    }
    static int dirCounter = 1;
    static int indentLevel = -1;
}
}
*****

```



**Hình 1-4: Kết xuất việc rảo xem các thư mục con.**

Chương trình bắt đầu nhận diện một thư mục (D:\THIEN\SACH\_C#) rồi tạo một đối tượng **DirectoryInfo** đối với thư mục này. Sau đó triệu gọi hàm **ExploreDirectory**, trao đối tượng **DirectoryInfo** cho hàm như là một thông số. Hàm **ExploreDirectory** lo hiển thị thông tin liên quan đến thư mục rồi lục tìm tất cả các thư mục con.

Danh sách tất cả các thư mục con của thư mục hiện hành sẽ nhận được bằng cách triệu gọi hàm **GetDirectories()**. Hàm này trả về một bản dãy các đối tượng

**DirectoryInfo.** Hàm **ExploreDirectory** là một hàm đệ quy; mỗi đối tượng **DirectoryInfo** được trao cho hàm **ExploreDirectory** theo phiên. Kết quả là ẩn lẩn xuống một cách đệ quy vào từng thư mục rồi bật lên khảo sát các thư mục con anh-em cho tới khi tất cả các thư mục đều được hiển thị. Khi nào hàm **ExploreDirectory** cuối cùng trở về, thì kết quả thống kê sẽ được in ra.

## 1.1.5 Tạo thư mục con thông qua lớp DirectoryInfo

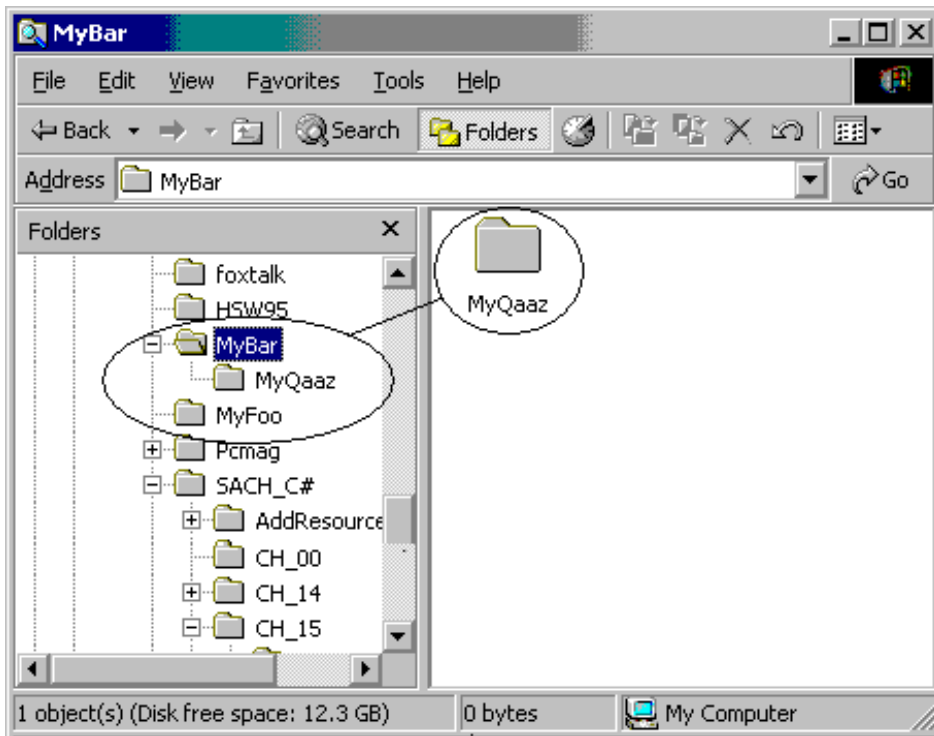
Bạn có thể nói rộng cấu trúc một thư mục bằng cách sử dụng đến hàm hành sự **CreateSubdirectory()**. Hàm này có thể tạo một thư mục đơn lẻ trên thư mục gốc, cũng như nhiều thư mục con nằm lồng nhau. Để minh hoạ, thí dụ 1-4 sau đây là đoạn mã nói rộng thư mục “D:\THIEN” với vài thư mục con custom.

### Thí dụ 1-4: Tạo thư mục con

```
*****
using System;
using System.IO;
namespace Prog_CSharp
{
    class MyDirectory
    {
        static void Main(string[] args)
        {
            MyDirectory t = new MyDirectory();
            DirectoryInfo dir = new DirectoryInfo(@"D:\THIEN");

            // Tạo những thư mục con trên D:\THIEN
            try
            {
                // thêm C:\THIEN\MyFoo
                dir.CreateSubdirectory("MyFoo");
                // thêm C:\THIEN\MyBar\MyQaaz
                dir.CreateSubdirectory(@"MyBar\MyQaaz");
            }
            catch(IOException e)
            {
                Console.WriteLine(e.Message);
            }
            Console.ReadLine();
        }
    }
}
*****
```

Nếu bạn nhìn thư mục D:\THIEN với Windows Explorer, bạn sẽ thấy các thư mục con mới thêm vào, chỗ chúng tôi cho khoanh tròn, hình 1-5:



**Hình 1-5 : Tạo thư mục con**

Mặc dù không nhất thiết cần hứng trị trả về của hàm hành sự **CreateSubdirectory()** nhưng bạn nên biết là một đối tượng **DirectoryInfo** sẽ được trả về nếu việc thi hành thành công, như theo sau đây:

```
// CreateSubdirectory() trả về một đối tượng DirectoryInfo
// tượng trưng cho item mới
try
{
    DirectoryInfo d = dir.CreateSubdirectory("MyFoo");
    Console.WriteLine("Created: {0}", d.FullName);

    d = dir.CreateSubdirectory(@"MyBar\MyQaaz");
    Console.WriteLine("Created: {0}", d.FullName);
}
catch (IOException e)
{
    Console.WriteLine(e.Message);
}
```

Bạn xem kết xuất ở hình 1-6

## 1.1.6 Các thành viên static của lớp Directory

Bạn đã biết lớp **DirectoryInfo** hành động ra sao. Bây giờ ta xem qua lớp **Directory**. Nhìn chung, lớp **Directory** bắt chước các chức năng của lớp **DirectoryInfo**, với một vài ngoại lệ, trong ấy có **GetLogicalDrives()**. Chúng tôi đề nghị bạn tham khảo trên MSDN để biết qua chi tiết của lớp này.

Trong lớp MyDirectory của chúng tôi phần còn lại sẽ liệt kê tên tất cả các ổ đĩa trong máy tính hiện hành và dùng hàm hành sự static **Delete()** để xóa đi những thư mục mà chúng tôi đã tạo ra trước đó \MyFoo và \MyBar\MyQaaz. Thí dụ 1-5 và hình 1-6 cho thấy bạn làm việc với các thành viên static của lớp Directory thế nào:

### Thí dụ 1-5: Làm việc với các thành viên static của lớp Directory

```
*****
using System;
using System.IO;
namespace Prog_CSharp
{
    class MyDirectory
    {
        static void Main(string[] args)
        {
            MyDirectory t = new MyDirectory();
            DirectoryInfo dir = new DirectoryInfo(@"D:\THIEN");

            try
            {
                DirectoryInfo d = dir.CreateSubdirectory("MyFoo");
                Console.WriteLine("Created: {0}", d.FullName);
                d = dir.CreateSubdirectory(@"MyBar\MyQaaz");
                Console.WriteLine("Created: {0}", d.FullName);
            }
            catch(IOException e)
            {
                Console.WriteLine(e.Message);
            }

            // liệt kê tất cả các tên ổ đĩa
            string[] drives = Directory.GetLogicalDrives();
            Console.WriteLine("Here are your drives:");
            foreach(string s in drives)
            {
                Console.WriteLine("->{0}", s);
            }

            // gỡ bỏ các thư mục bạn tạo ra trước đó
            Console.Write("Going to delete\n->" + dir.FullName +
                "\\\MyBar\\MyQaaz.\nand\n->" + dir.FullName +
                "\\\MyFoo.\n" + "Press a key to continue!");
            Console.Read();
        }
    }
}
```





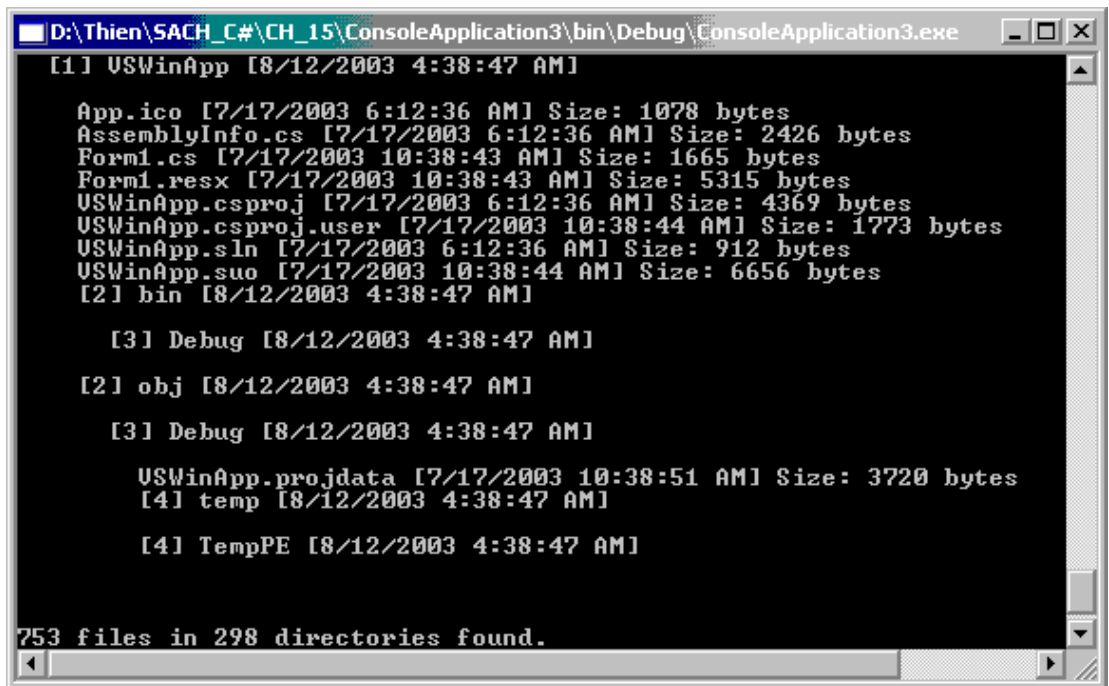
Vai trò của lớp **FileInfo** là bao trọn một số chi tiết liên quan đến các tập tin hiện hữu trên ổ đĩa (như thời gian tạo lập, kích thước, các thuộc tính v.v...). Ngoài những lô chức năng được thừa hưởng từ lớp **FileSystemInfo**, lớp **FileInfo** cũng có một số thành viên cốt lõi mà chúng tôi sẽ cho liệt kê trên bảng 1-5 sau đây:

**Bảng 1-5: Các thành viên cốt lõi của lớp FileInfo**

Tên thành viên	Ý nghĩa
<b>AppendText()</b>	Tạo một kiểu dữ liệu <b>StreamWriter</b> (sẽ được mô tả sau) cho ghi nối đuôi phần văn bản lên một tập tin được khai báo.
<b>CopyTo()</b>	Chép một tập tin hiện hữu lên một tập tin mới.
<b>Create()</b>	Tạo mới một tập tin và trả về một đối tượng <b>FileStream</b> (sẽ được mô tả sau) để tương tác với tập tin được tạo.
<b>CreateText()</b>	Tạo một đối tượng <b>StreamWriter</b> viết một tập tin văn bản lên một tập tin được khai báo.
<b>Delete()</b>	Cho gỡ bỏ một tập tin được khai báo dựa trên một thẻ hiện <b>FileInfo</b> .
<b>Directory</b>	Đi lấy một thẻ hiện của thư mục cha-mẹ.
<b>DirectoryName</b>	Đi lấy một lối tìm về trọn vẹn chỉ về một tập tin.
<b>Exists()</b>	Trả về true nếu tập tin hiện hữu.
<b>GetAttributes()</b>	Đi lấy hoặc đặt để <b>FileAttributes</b> đối với tập tin được khai báo.
<b>SetAttributes()</b>	Xem bảng 1-4 để biết các trị của enum <b>FileAttributes</b> .
<b>GetCreationTime()</b>	Đi lấy hoặc đặt để thời gian tạo tập tin.
<b>SetCreationTime()</b>	
<b>GetLastAccessTime()</b>	Đi lấy hoặc đặt để thời gian truy xuất tập tin lần chót nhất.
<b>SetLastAccessTime()</b>	
<b>GetLastWriteTime()</b>	Đi lấy hoặc đặt để thời gian viết lên tập tin lần chót nhất.
<b>SetLastWriteTime()</b>	
<b>Length</b>	Đi lấy kích thước của tập tin hoặc thư mục không dùng đến nữa.
<b>MoveTo()</b>	Di chuyển một tập tin được khai báo về chỗ ở mới; có mục chọn cho biết tên tập tin mới.
<b>Name</b>	Đi lấy tên tập tin.
<b>Open()</b>	Cho mở một tập tin với những quyền khác nhau về đọc/viết/chia sẻ sử dụng.
<b>OpenRead()</b>	Hàm hành sự public static cho mở một <b>FileStream</b> read-only lên tập tin.
<b>OpenText()</b>	Tạo một đối tượng <b>StreamReader</b> (sẽ được mô tả sau) lo đọc từ một tập tin văn bản hiện hữu.
<b>OpenWrite()</b>	Tạo một đối tượng <b>FileStream</b> đọc/viết theo một lối tìm về được khai báo.

### 1.1.7.1 Rảo xem các tập tin và thư mục

Thí dụ 1-6 thay đổi đôi chút thí dụ 1-3, có thêm đoạn mã để đi lấy đối tượng **FileInfo** cho mỗi tập tin trên mỗi thư mục con. Đối tượng **FileInfo** này dùng hiển thị tên tập tin (**Name**) kèm theo kích thước tập tin (**Length**) cũng như thời gian viết lên lần chót (**LastWriteTime**). Sau đây là bảng liệt in, với các dòng in đậm là đoạn mã thêm vào. Hình 1-7 cho thấy kết xuất của chương trình.



Hình 1-7 : Rảo xem các tập tin và thư mục (trích một đoạn)

#### Thí dụ 1-6: Rảo xem các tập tin và thư mục con

```

*****
using System;
using System.IO;
namespace Prog_CSharp
{
    class MyDirectory
    {
        static void Main(string[] args)
        {
            MyDirectory t = new MyDirectory();
            // chọn thư mục con ban đầu
            string theDirectory = @"D:\THIEN\SACH_C#";

```

```

        // triệu gọi hàm khảo sát thư mục, hiển thị ngày
        // truy xuất tất cả các thư mục con
        DirectoryInfo dir = new DirectoryInfo(theDirectory);
        t.ExploreDirectory(dir);
        // Xong! In ra thống kê
        Console.WriteLine("\n\n{0} files in
                           {1} directories found.\n", fileCounter, dirCounter);
    }

    // Hàm đệ quy đối với mỗi thư mục
    private void ExploreDirectory(DirectoryInfo dir)
    {
        indentLevel++; // push a directory level
        // tạo indentation cho thư mục con
        for (int i = 0; i < indentLevel; i++)
        { Console.Write(" "); // hai space cho mỗi level
        }

        // in ra thư mục và thời gian truy xuất chót nhất
        Console.WriteLine("[{0}] {1} [{2}]\n",
                           indentLevel, dir.Name, dir.LastAccessTime);

        // đi lấy tất cả các tập tin trên thư mục rồi in ra
        // tên, kích thước v.v..
        FileInfo[] filesDir = dir.GetFiles();
        foreach(FileInfo file in filesDir)
        { // canh thụt thêm cho tập tin nằm dưới thư mục
            for (int i = 0; i < indentLevel+1; i++)
                Console.Write(" ");
            Console.WriteLine("{0} [{1}] Size: {2} bytes",
                               file.Name, file.LastWriteTime, file.Length);
            fileCounter++;
        }

        // đi lấy tất cả các thư mục trên thư mục hiện hành
        // rồi triệu gọi hàm này một cách đệ quy
        DirectoryInfo[] directories = dir.GetDirectories();
        foreach(DirectoryInfo newDir in directories)
        { dirCounter++;
            ExploreDirectory(newDir);
        }
        indentLevel--;
    }

    static int dirCounter = 1;
    static int indentLevel = -1;
    static int fileCounter = 0;
}
}

```

\*\*\*\*\*

### 1.1.7.2 Tạo một tập tin vật lý

Điều đầu tiên mà bạn có thể nhận thấy trên bảng 1-5, là nhiều hàm hành sự được định nghĩa bởi **FileInfo** trả về một kiểu dữ liệu đặc thù (**FileStream**, **StreamReader**, **StreamWriter** v.v..) cho phép bạn bắt đầu đọc và viết dữ liệu lên (hoặc từ) tập tin liên hệ theo nhiều cách khác nhau. Chương này sẽ xem xét đến những kiểu dữ liệu mới. Thí dụ 1-7 cho thấy lớp FileManipulator sau đây minh họa cách chung chung nhất (và ít uyển chuyển) để tạo một tập tin theo chương trình:

#### Thí dụ 1-7: Tạo một tập tin vật lý

```
*****
public class FileManipulator
{
    public static int Main(string[] args)
    { // Tạo một tập tin mới trên C:\
        FileInfo f = new FileInfo(@"D:\THIEN\SACH_C#\Test.txt");
        FileStream fs = f.Create();

        // In ra vài đặc tính cơ bản của tập tin test.txt
        Console.WriteLine("Creation: {0}", f.CreationTime);
        Console.WriteLine("Full name: {0}", f.FullName);
        Console.WriteLine("Full atts: {0}", f.Attributes.ToString());
        Console.WriteLine("Press a key to delete file");
        Console.Read();

        // đóng lại file stream và xóa tập tin
        fs.Close();
        f.Delete();
        return 0;
    }
}
*****
```

Bạn để ý là hàm hành sự **Create()** trả về một đối tượng **FileStream** cho phép bạn đóng lại tập tin mới trước khi gỡ bỏ khỏi ổ đĩa (bạn sẽ thấy về sau những trường hợp sử dụng khác của **FileStream**). Khi bạn cho chạy chương trình này bạn có thể thấy phần kết xuất kên màn hình và tập tin mới được tạo ra trên thư mục được khai báo hiển thị thông qua Windows Explorer như hình 1-8.

### 1.1.7.3 Khảo sát hàm hành sự *FileInfo.Open()*

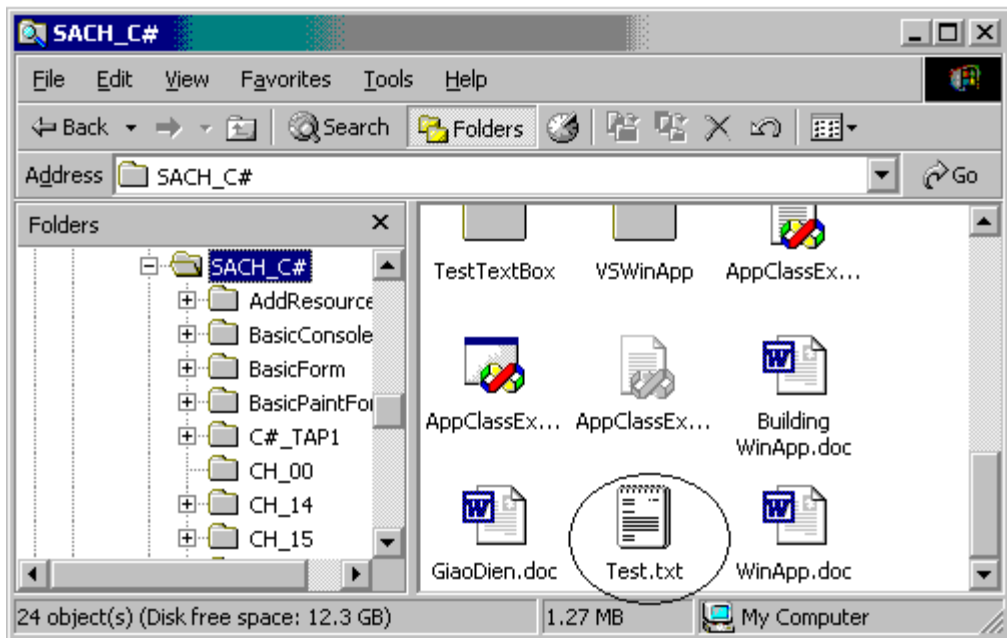
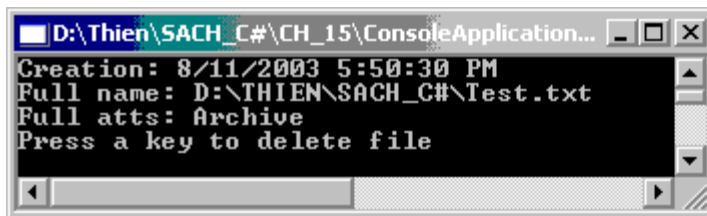
Hàm **Open()** của lớp **FileInfo** có thể dùng mở các tập tin hiện hữu cũng như tạo những tập tin mới với nhiều chính xác hơn là hàm hành sự **FileInfo.Create()**. Thí dụ 1-8 sau đây minh họa điều kể trên:

**Thí dụ 1-8: Mở (hoặc tạo) một tập tin vật lý**

```

*****
...
// Mở (hoặc tạo) một tập tin với quyền read/write (no sharing) rồi trử
// mục quản tập tin (file handle) lên một đối tượng FileStream
FileInfo f2 = new FileInfo(@"C:\HelloThere.ini");
FileStream s = f2.Open(FileMode.OpenOrCreate, FileAccess.ReadWrite,
    FileShare.None);
s.Close();
f2.Delete();
*****

```



**Hình 1-8 : Tạo một tập tin vật lý bằng chương trình.**

### 1.1.7.3 Khảo sát hàm hành sự *FileInfo.Open()*

Phiên bản nạp chồng của hàm hành sự **Open()** cần đến 3 thông số: thông số thứ nhất cho biết các mục chọn mở (nghĩa là tạo một tập tin mới, mở một tập tin hiện hữu, kết nối

đuôi một tập tin v.v..) được khai báo thông qua enumeration **FileMode**, như theo bảng 1-6 sau đây:

**Bảng 1-6: Các trị của Enumeration FileMode**

Tên thành viên	Ý nghĩa
<b>Append</b>	Cho mở tập tin nếu hiện hữu và đi tìm đầu cuối tập tin. Nếu tập tin không hiện hữu thì cho tạo mới một tập tin. Nên nhớ, <b>FileMode.Append</b> chỉ có thể dùng phối hợp với <b>FileAccess.Write</b> . Nếu cố đọc tập tin thì sẽ gây ra biệt lệ <b>ArgumentException</b> .
<b>Create</b>	Cho biết <i>hệ điều hành</i> phải tạo một tập tin mới. Nếu tập tin hiện hữu, thì nó sẽ bị viết đè chồng lên. Hàm này đòi hỏi phải có <b>FileIOPermissionAccess.Write</b> và <b>FileIOPermissionAccess.Append</b> . Hàm <b>System.IO.FileMode.Create</b> tương đương yêu cầu là nếu tập tin không hiện hữu, sử dụng đến hàm <b>CreateNew</b> ; bằng không sử dụng đến hàm <b>Truncate</b> .
<b>CreateNew</b>	Cho biết <i>hệ điều hành</i> phải tạo một tập tin mới. Hàm này đòi hỏi phải có quyền đọc <b>FileIOPermissionAccess.Read</b> và quyền ghi nội đuôi <b>FileIOPermissionAccess.Append</b> . Nếu tập tin đã hiện hữu, biệt lệ <b>IOException</b> sẽ gây ra.
<b>Open</b>	Cho biết <i>hệ điều hành</i> phải mở một tập tin hiện hữu. Hàm này đòi hỏi phải có quyền đọc <b>FileIOPermissionAccess.Read</b> . Một biệt lệ sẽ được tung ra <b>System.IO.FileNotFoundException</b> nếu tập tin không hiện hữu.
<b>OpenOrCreate</b>	Cho biết <i>hệ điều hành</i> phải mở một tập tin nếu nó hiện hữu, bằng không một tập tin mới sẽ được tạo ra. Nếu tập tin được mở với truy xuất đọc <b>FileAccess.Read</b> , thì đòi hỏi phải có quyền đọc <b>FileIOPermissionAccess.Read</b> . Nếu truy xuất tập tin là <b>FileAccess.ReadWrite</b> và nếu tập tin hiện hữu, thì đòi hỏi phải có quyền viết <b>FileIOPermissionAccess.Write</b> . Nếu truy xuất tập tin là <b>FileAccess.ReadWrite</b> và tập tin không hiện hữu, thì đòi hỏi phải có quyền <b>FileIOPermissionAccess.Append</b> ngoài <b>Read</b> và <b>Write</b> .
<b>Truncate</b>	Cho biết <i>hệ điều hành</i> phải mở một tập tin hiện hữu. Một khi đã được mở, tập tin phải bị xén đi làm thế nào kích thước bằng zero. Hàm này đòi hỏi phải có quyền truy xuất viết <b>FileIOPermissionAccess.Write</b> . Khi cố đọc một tập tin được mở với hàm <b>Truncate</b> sẽ gây ra một biệt lệ.

Thông số thứ hai, **FileAccess**, dùng xác định hành vi read/write của dòng dữ liệu nằm dưới như bảng 1-7:

**Bảng 1-7: Các trị của Enumeration FileAccess**

Tên thành viên	Ý nghĩa
<b>Read</b>	Cho biết chỉ cho đọc tập tin mà thôi. Dữ liệu có thể đọc từ tập tin. Phối hợp với <b>Write</b> để có quyền truy xuất read/write.
<b>ReadWrite</b>	Cho biết quyền truy xuất đọc và viết lên một tập tin. Dữ liệu có thể đọc xuống hoặc viết lên một tập tin.
<b>Write</b>	Cho biết quyền truy xuất viết lên một tập tin. Dữ liệu có thể viết lên tập tin. Có thể phối hợp với <b>Read</b> để có quyền truy xuất read/write.

Cuối cùng, thông số thứ ba, **FileShare**, cho biết tập tin có được chia sẻ sử dụng với các tập tin khác hay không. Bảng 1-8 cho thấy các trị của Enumeration **FileShare**:

**Bảng 1-8: Các trị của Enumeration FileShare**

Tên thành viên	Ý nghĩa
<b>None</b>	Từ chối chia sẻ sử dụng tập tin hiện hành. Bất cứ yêu cầu nào mở tập tin này (bởi tiến trình này hoặc tiến trình khác) đều thất bại cho tới khi tập tin được đóng lại.
<b>Read</b>	Cho phép lệnh đi sau mở tập tin để đọc. Nếu flag này không được khai báo, thì bất cứ yêu cầu nào mở tập tin này để đọc (bởi tiến trình này hoặc tiến trình khác) đều thất bại cho tới khi tập tin được đóng lại. Tuy nhiên, nếu flag này được khai báo, vẫn phải cần có những quyền bổ sung để truy xuất tập tin.
<b>ReadWrite</b>	Cho phép lệnh đi sau mở tập tin để đọc hoặc viết. Nếu flag này không được khai báo, thì bất cứ yêu cầu nào mở tập tin này để đọc hoặc viết (bởi tiến trình này hoặc tiến trình khác) đều thất bại cho tới khi tập tin được đóng lại. Tuy nhiên, nếu flag này được khai báo, vẫn cần có những quyền bổ sung để truy xuất tập tin.
<b>Write</b>	Cho phép lệnh đi sau mở tập tin để viết. Nếu flag này không được khai báo, thì bất cứ yêu cầu nào mở tập tin này để viết (bởi tiến trình này hoặc tiến trình khác) đều thất bại cho tới khi tập tin được đóng lại. Tuy nhiên, nếu flag này được khai báo, vẫn cần có những quyền bổ sung để truy xuất tập tin.



### 1.1.7.4 Các hàm *FileInfo.OpenRead()* và *FileInfo.OpenWrite()*

Ngoài hàm **Open()** kể trên, lớp **FileInfo** còn có hai hàm **OpenRead()** và **OpenWrite()** trả về đối tượng **FileStream** chỉ đọc mà thôi (read-only) hoặc chỉ viết mà thôi (write-only). Sau đây là một thí dụ:

```
// Đi lấy một đối tượng FileStream với quyền chỉ đọc mà thôi
FileInfo f3 = new FileInfo(@"C:\boot.ini");
FileStream readOnlyStream = f3.OpenRead();
...
readOnlyStream.Close();

// Bây giờ đi lấy một đối tượng FileStream với
// quyền chỉ được viết mà thôi
FileInfo f4 = new FileInfo(@"C:\config.sys");
FileStream writeOnlyStream = f4.OpenWrite();
...
writeOnlyStream.Close();
```

### 1.1.7.5 Các hàm *OpenText()*, *CreateText()* và *AppendText()* của *FileInfo*

Một hàm khác chuyên mở tập tin là **FileInfo.OpenText()**. Khác với **Open()**, **OpenRead()** và **OpenWrite()**, hàm **OpenText()** trả về một đối tượng kiểu **StreamReader**, thay vì kiểu **FileStream**. Sau đây là một số thí dụ con:

```
// Đi lấy một đối tượng StreamReader
FileInfo f5 = new FileInfo(@"C:\bootlog.txt");
StreamReader sreader = f5.OpenText();
sreader.Close();
```

Hai hàm chót là **CreateText()** (tạo một đối tượng **StreamWriter** viết ra mới một tập tin văn bản) và **AppendText()** (tạo một đối tượng **StreamWriter** ghi nối đuôi văn bản vào một tập tin được tượng trưng bởi thể hiện **FileInfo** này), cả hai trả về đối tượng **StreamWriter**. Sau đây là một số thí dụ con:

```
// Đi lấy vài đối tượng StreamWriter
FileInfo f6 = new FileInfo(@"C:\AnotherText.txt");
f6.Open(FileMode.Create, FileAccess.ReadWrite);
if (!f6.Exists)
{
    StreamWriter swriter = f6.CreateText();
    swriter.WriteLine("Hello");
    swriter.WriteLine("And");
}
```

```

        swriter.WriteLine("Welcome");
        swriter.Close();
    }

    FileInfo f7 = new FileInfo(@"C:\FinalText.txt");
    f7.Open(FileMode.Create, FileAccess.ReadWrite);
    if (!f7.Exists)
    {
        StreamWriter swriter = f7.CreateText();
        swriter.WriteLine("Hello");
        swriter.WriteLine("And");
        swriter.WriteLine("Welcome");
        swriter.Close();
    }
    else
    {
        StreamWriter swriterAppend = f7.AppendText();
        swriterAppend.WriteLine("This");
        swriterAppend.WriteLine("is Extra");
        swriterAppend.WriteLine("Text");
        swriterAppend.Close();
    }
}

```

Tới đây coi như bạn đã hiểu sơ qua các chức năng của lớp **FileInfo**. Còn bạn làm gì với các đối tượng **FileStream**, **StreamReader** và **StreamWriter** thì hạ hồi phân giải. Bạn để ý là lớp **File** cũng cung cấp những chức năng tương tự thông qua những thành viên static. Bạn sẽ thấy cách dùng lớp **File** thế nào vào những lúc thích ứng. Bạn có thể tham khảo MSDN liên quan đến các thành viên của lớp này.

### 1.1.7.6 Thay đổi các tập tin

Như bạn có thể thấy ở trên, ta có thể dùng lớp **FileInfo** để tạo, sao chép, đổi tên, và gỡ bỏ các tập tin. Thí dụ 1-9 kế tiếp sẽ tạo một thư mục con, chép các tập tin lên thư mục này, cho đổi tên vài tập tin, gỡ bỏ một số và cuối cùng xóa toàn bộ thư mục.

**Bạn để ý:** Để dàn dựng thí dụ này, bạn cho tạo một thư mục **\test** rồi chép thư mục **media** từ WinNT vào thư mục **test**. không được làm việc trực tiếp trên các tập tin của WinNT; khi làm việc với các tập tin hệ thống bạn phải hết sức thận trọng.

Bước đầu tiên là tạo một đối tượng **DirectoryInfo** để trắc nghiệm thư mục:

```

string theDirectory = @"D:\Test\Media";
DirectoryInfo dir = new DirectoryInfo(theDirectory);

```

Tiếp theo, là tạo một thư mục con trong lòng thư mục trắc nghiệm bằng cách triệu gọi hàm **CreateSubDirectory()** trên đối tượng **DirectoryInfo**. Bạn sẽ nhận trả về một đối tượng **DirectoryInfo** mới, tượng trưng cho một thư mục mới được tạo:

```
string newDirectory = "newTest";  
DirectoryInfo newSubDir = dir.CreateSubDirectory(newDirectory);
```

Bây giờ bạn có thể rảo qua thư mục test và chép các tập tin từ đây lên thư mục newTest mới được tạo ra.

```
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);  
}
```

Bạn để ý cú pháp của hàm hành sự **CopyTo()**. Đây là một hàm hành sự của đối tượng **FileInfo**. Bạn trao qua lỗi tìm về trọn vẹn của tập tin mới bao gồm tên trọn vẹn và extension.

Một khi bạn đã chép các tập tin, bạn có thể lấy danh sách các tập tin trên thư mục con mới và làm việc trực tiếp với các tập tin này.

```
filesInDir = newSubDir.GetFiles();  
foreach (FileInfo file in filesInDir)  
{
```

Bạn tạo một biến số nguyên đơn giản cho mang tên **counter** và dùng biến này để đổi tên các tập tin:

```
if (counter++ % 2 == 0)  
{  
    file.MoveTo(fullName + ".bak");  
    Console.WriteLine("{0} renames to {1}", fullName, file.FullName);  
}
```

Bạn đổi tên một tập tin bằng cách “di chuyển” (moving) nó lên cùng thư mục nhưng với tên mới. Lẽ dĩ nhiên, bạn cũng có thể di chuyển tập tin lên một thư mục mới với tên nguyên thủy hoặc bạn có thể di chuyển và đổi tên cùng một lúc.

Bạn cho đổi tên đối với tập tin khác, và gỡ bỏ những tập tin bạn không đổi tên.

```
file.Delete();  
Console.WriteLine("{0} deleted.", fullName);
```

Một khi bạn thao tác xong trên tất cả các tập tin, bạn có thể cho xóa sạch bằng cách cho gỡ bỏ toàn bộ thư mục con:

```
newSubDir.Delete(true);
```

Thông số bool trong hàm **Delete()** cho biết liệu xem có phải là một xóa sổ đệ quy hay không. Nếu bạn trả false, và nếu thư mục có thư mục con với các tập tin trên ấy, thì một biệt lệ sẽ được tung ra.

Thí dụ 1-7 là toàn bộ chương trình, và hình 1-9 là phần kết xuất (trích một đoạn)

### ***Thí dụ 1-7: Tạo một thư mục con và thao tác trên các tập tin***

\*\*\*\*\*

```
using System;
using System.IO;
namespace Prog_CSharp
{
    class MyDirectory
    {
        static void Main(string[] args)
        {
            MyDirectory t = new MyDirectory();
            // chọn thư mục con ban đầu
            string theDirectory = @"D:\Test\Media";
            // triệu gọi hàm khảo sát thư mục, hiển thị ngày
            // truy xuất và tất cả các thư mục con
            DirectoryInfo dir = new DirectoryInfo(theDirectory);
            t.ExploreDirectory(dir);
        }

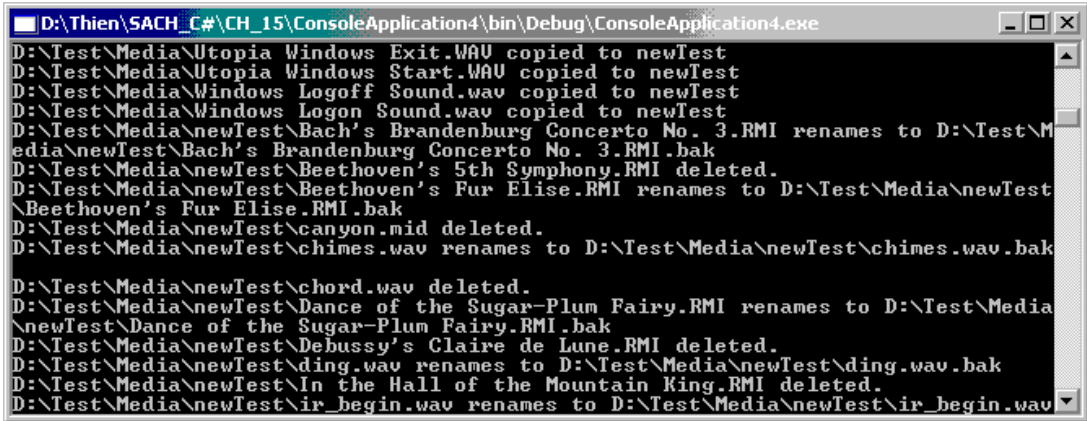
        // Cho nó chạy với một tên thư mục
        private void ExploreDirectory(DirectoryInfo dir)
        {
            // tạo một thư mục mới
            string newDirectory = "newTest";
            DirectoryInfo newSubDir =
                dir.CreateSubDirectory(newDirectory);
            // đi lấy tất cả các tập tin trên thư mục
            // chép về 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);
            }

            // đi lấy collection các tập tin
            filesInDir = newSubDir.GetFiles();

            // gỡ bỏ một số và đổi tên một số
            int counter = 0;
            foreach (FileInfo file in filesInDir)
            {
                string fullName = file.FullName;
                if (counter++ % 2 == 0)
                {
                    file.MoveTo(fullName + ".bak");
                    Console.WriteLine("{0} renames to {1}",
                        fullName, file.FullName);
                }
            }
        }
    }
}
```

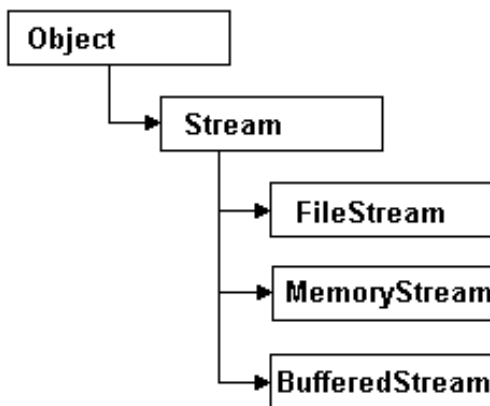
```
        else
        {
            file.Delete();
            Console.WriteLine("{0} deleted.", fullName);
        }
    }
    newSubDir.Delete(true);
}
}
```

\*\*\*\*\*



**Hình 1-9: Thao tác lên thư mục và các tập tin**

## 1.2 Đọc và viết dữ liệu



Hình 15-10: Các lớp dẫn xuất từ lớp Stream

của **Stream** tương trưng dữ liệu như là một dòng dữ liệu thô dạng bytes (thay vì dữ liệu

Đọc và viết dữ liệu sẽ được thực hiện thông qua lớp **Stream**. Stream là dòng dữ liệu chảy đi. Đây là một thực thể (entity) có khả năng nhận được hoặc tạo ra một “nhúm” dữ liệu. **System.IO.Stream** là một lớp abstract định nghĩa một số thành viên chịu hỗ trợ việc đọc/viết đồng bộ (synchronous) hoặc không đồng bộ (asynchronous) đối với khối trữ tin (nghĩa là một tập tin trên đĩa hoặc tập tin trên ký ức). Hình 1-10 cho thấy cây đẳng cấp của lớp **Stream**.

Vì **Stream** là một lớp abstract, nên bạn chỉ có thể làm việc với những lớp được dẫn xuất từ **Stream**. Các hâu duê

dạng văn bản). Ngoài ra, các lớp được dẫn xuất từ **Stream** hỗ trợ việc truy tìm (seek) nghĩa là một tiến trình nhận lấy và điều chỉnh vị trí trên một dòng chảy. Trước khi tìm hiểu những chức năng mà lớp **Stream** cung cấp, bạn nên xem qua các thành viên của lớp **Stream**, bảng 1-9.

**Bảng 1-9: Các thành viên của lớp abstract Stream**

Tên thành viên	Ý nghĩa
<b>BeginRead()</b>	Bắt đầu một tác vụ đọc (hoặc viết) bất đồng bộ (asynchronous).
<b>BeginWrite()</b>	
<b>CanRead</b>	Xác định liệu xem stream hiện hành chịu hỗ trợ read, seek và/hoặc
<b>CanSeek</b>	write hay không.
<b>CanWrite</b>	
<b>Close()</b>	Cho đóng lại stream hiện hành và giải phóng bất cứ nguồn lực nào (chẳng hạn socket, handle) được gắn liền với stream hiện hành.
<b>Flush()</b>	Cho nhật tu nguồn dữ liệu nằm đằng sau với tình trạng hiện hành của vùng đệm rồi cho xóa đi vùng đệm. Nếu một stream không thiết lập một vùng đệm thì hàm này sẽ không làm gì cả.
<b>Length</b>	Trả về kích thước của stream tính theo bytes.
<b>Position</b>	Xác định vị trí trên stream hiện hành.
<b>Read()</b>	Cho đọc một loạt bytes (hoặc chỉ một byte) từ stream hiện hành rồi
<b>ReadByte()</b>	cho nhảy vị trí hiện hành trên stream về số byte đã đọc vào.
<b>Seek()</b>	Cho đặt để vị trí trên stream hiện hành.
<b>SetLength()</b>	Cho đặt để chiều dài của stream hiện hành.
<b>Write()</b>	Viết một loạt byte (hoặc chỉ một byte) lên stream hiện hành rồi cho
<b>WriteByte()</b>	nhảy vị trí hiện hành trên stream về số byte đã viết lên.

## 1.2.1 Làm việc với các tập tin nhị phân

Chúng tôi sẽ bắt đầu dùng lớp cơ bản **Stream** để thực hiện việc *đọc nhị phân* (binary read) một tập tin. Từ *binary read* dùng để phân biệt với việc *đọc văn bản* (text read). Nếu bạn không biết chắc một tập tin thuộc loại text thì cách an toàn nhất là xem nó như là một dòng bytes được biết dưới cái tên là một *tập tin nhị phân* (binary file).

Các hàm quan trọng trong lớp **Stream** này là **Read()**, **Write()**, **BeginRead()**, **BeginWrite()**, và **Flush()**. Chúng tôi sẽ duyệt qua các hàm này trong phần này.

Muốn thực hiện một binary read, bạn bắt đầu tạo một cặp đối tượng **Stream**, một để đọc và một để viết.

```
Stream inputStream = File.OpenRead(@"C:\test\source\test1.cs");  
Stream outputStream = File.OpenWrite(@"C:\test\source\test1.bak");
```

Muốn mở các tập tin để đọc hoặc viết, bạn dùng đến hàm static của lớp **File**, **OpenRead()** và **OpenWrite()**. Phiên bản static nạp chồng của các hàm trên chỉ lấy một thông số là lối tìm về (path) đối với tập tin, như bạn đã thấy trên hai câu lệnh trên.

Đọc nhị phân làm việc thông qua một vùng đệm (buffer). Vùng đệm đơn giản là một bản dãy các bytes lo việc cầm giữ dữ liệu đọc vào bởi hàm **Read()**.

Bạn đi vào buffer, dựa theo di số (offset) theo dãy dữ liệu được bắt đầu trữ khi đọc vào, và số bytes phải đọc. Hàm **inputStream.Read()** sẽ đọc từ “kho trữ dữ liệu” (data store) ghi lên buffer rồi trả về số bytes đọc được. Nó cứ thế mà tiếp tục cho tới khi nào không còn dữ liệu để đọc.

```
while ((bytesRead = inputStream.Read(buffer, 0, SIZE_BUFF)) > 0)  
{  
    outputStream.Write(buffer, 0, bytesRead);  
}
```

Mỗi lần buffer đầy thì viết lên tập tin xuất. Các đối mục của hàm **Write** là buffer, offset trên buffer bắt đầu đọc, và số bytes phải viết. Bạn để ý là bạn viết đúng số bytes bạn vừa đọc vào.

Thí dụ 1-8 cho thấy toàn bộ chương trình.

### ***Thí dụ 1-8: Thi công một binary read và write lên một tập tin***

\*\*\*\*\*

```
namespace Progr_CSharp  
{  
    using System;  
    using System.IO;  
  
    class Tester  
    {  
        const int SizeBuff = 1024; // kích thước vùng đệm  
  
        public static void Main()  
        {  
            Tester t = new Tester();  
            t.Run();  
        }  
        // Cho chạy với tên thư mục  
        private void Run()  
        {  
            // tập tin đọc vào và viết ra  
            Stream inputStream = File.OpenRead(@"C:\test\source\test1.cs");  
            Stream outputStream = File.OpenWrite(  

```

```

                                @"C:\test\source\test1.bak");
    // tạo một buffer và biến cho biết số byte
    byte[] buffer = new Byte[SizeBuffer];
    int bytesRead;
    // vừa đọc vừa viết
    while ((bytesRead = inputStream.Read(buffer, 0, SizeBuff)) > 0)
    {   outputStream.Write(buffer, 0, bytesRead);
    }
    // dọn dẹp trước khi ra về
    inputStream.Close();
    outputStream.Close();
}
}
}
*****

```

Kết quả chạy chương trình này là chép tập tin nhập (test1.cs) viết lên cùng thư mục qua một tập tin khác mang tên test1.bak.

## 1.2.2 Làm việc với FileStream

Lớp **FileStream** đem lại việc thi công cho những thành viên của lớp abstract **Stream** theo một thể thức thích hợp đối với file-base streaming. Giống như với các lớp **DirectoryInfo** và **FileInfo**, lớp **FileStream** cho phép mở những tập tin hiện hữu cũng như tạo mới tập tin. Khi tạo tập tin, lớp **FileStream** thường dùng những enum **FileMode**, **FileAccess** và **FileShare**. Thí dụ, lệnh sau đây tạo một tập tin mới (Test.dat) trên một thư mục ứng dụng thích hợp:

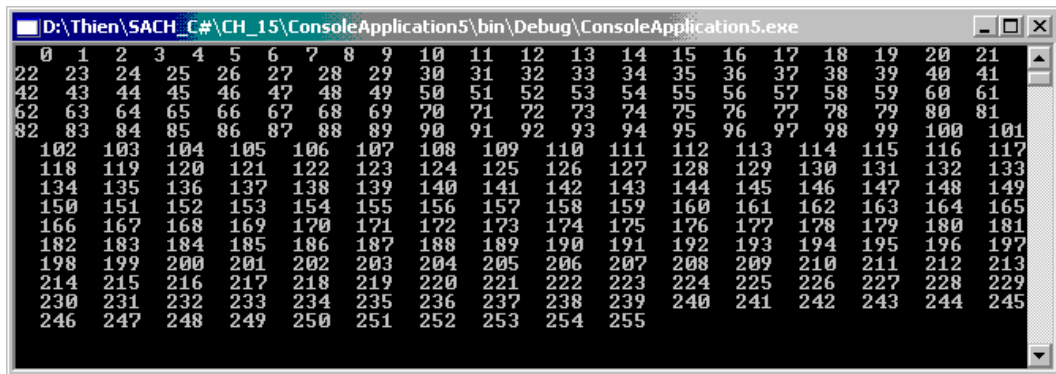
```

// tạo một tập tin mới trên thư mục làm việc
FileStream myFStream = new FileStream("test.dat",
                                     FileMode.OpenOrCreate, FileAccess.ReadWrite);

```

Bạn có thể thử nghiệm với khả năng read/write đồng bộ của lớp **FileStream**. Muốn viết một dòng byte lên một tập tin, bạn triệu gọi hàm dẫn xuất **WriteByte()** hoặc **Write()**, cả hai tự động cho nhảy tới một con trỏ nội tại. Còn muốn đọc lại từ một tập tin, thì dùng đến hàm **Read()** hoặc **ReadByte()**. Sau đây là thí dụ 1-9 với kết xuất lên màn hình (hình 1-11):





Hình 1-11 : Làm việc với FileStream

***Thí dụ 1-9: Viết lên một tập tin rồi đọc lại dùng lớp FileStream***

```

*****
// tạo một tập tin mới trên thư mục làm việc
FileStream myFStream = new FileStream(
    "test.dat", FileMode.OpenOrCreate, FileAccess.ReadWrite);
// viết byte lên tập tin *.dat
for (int i = 0; i < 256; i++)
{
    myFStream.WriteByte((byte)i);
}
// cho trở lại vị trí ban đầu
myFStream.Position = 0;
// đọc lại byte từ tập tin *.dat
for (int i = 0; i < 256; i++)
{
    Console.Write(myFStream.ReadByte());
}
// dọn dẹp
myFStream.Close();
*****

```

Nếu bạn mở tập tin mới này từ Visual Studio .NET IDE, bạn có thể thấy byte stream nằm đằng sau.

**1.2.3 Làm việc với MemoryStream**

Lớp **MemoryStream** hoạt động cũng tương tự như lớp **FileStream** với khác biệt hiển nhiên là giờ đây bạn viết lên ký ức thay vì lên một tập tin vật lý. Vì cả hai **MemoryStream** và **FileStream** đều dẫn xuất từ **Stream**, bạn có thể nhậ tu đoạn mã trước như sau:

***Thí dụ 1-10: Viết lên một tập tin rồi đọc lại bằng lớp MemoryStream***

```

*****
// tạo một tập tin mới trên thư mục làm việc
MemoryStream myMemStream = new MemoryStream();
myMemStream.Capacity = 256;
// viết byte lên tập tin *.dat
for (int i = 0; i < 256; i++)
{ myMemStream.WriteByte((byte)i);
}
// cho trở lại vị trí ban đầu
myMemStream.Position = 0;
// đọc lại byte từ tập tin *.dat
for (int i = 0; i < 256; i++)
{ Console.Write(myMemStream.ReadByte());
}
// dọn dẹp
myMemStream.Close();
*****

```

Bạn thấy thí dụ 1-10 không khác chi mấy so với thí dụ 1-9. Chỉ khác biệt là tập tin và ký ức. Ngoài những thành viên được kế thừa, lớp **MemoryStream** cung cấp một vài thành viên được liệt kê sau đây bởi bảng 1-10. Trong thí dụ 1-10, bạn thấy chúng tôi dùng đến thuộc tính **Capacity** cho biết dành bao nhiêu ký ức cho tác vụ:

### **Bảng 1-10: Các thành viên của lớp *MemoryStream***

<b>Tên thành viên</b>	<b>Ý nghĩa</b>
<b>Capacity</b>	Đi lấy hoặc đặt để số bytes sẽ được cấp phát cho stream này.
<b>GetBuffer()</b>	Hàm này trả về bản dãy các byte không dấu (unsigned) từ đầu stream được cấu tạo.
<b>ToArray()</b>	Cho viết toàn nội dung của stream lên một bản dãy byte không cần biết thuộc tính <b>Position</b> .
<b>WriteTo()</b>	Cho viết toàn nội dung của MemoryStream này lên một lớp dẫn xuất từ stream (chẳng hạn một tập tin).

Bạn để ý sự phối hợp giữa hai lớp **MemoryStream** và **FileStream**. Khi bạn sử dụng đến hàm hành sự **WriteTo()**, bạn có thể chuyển dễ dàng dữ liệu trữ trên ký ức tuôn ghi lên một tập tin. Ngoài ra, bạn cũng có thể tìm lại memory stream như là một bản dãy byte:

```

// Dump dữ liệu ký ức lên một tập tin
FileStream dumpFile = new FileStream("Dump.dat", FileMode.Create,
                                   FileAccess.ReadWrite);
myMemStream.WriteTo(dumpFile);
// Dump dữ liệu ký ức lên một bản dãy byte
byte[] bytesInMemory = myMemStream.ToArray();
// dọn dẹp
myMemStream.Close();

```

## 1.2.4 Làm việc với BufferedStream

Trong thí dụ 1-8, bạn đã tạo một vùng đệm (buffer) để trữ dữ liệu đọc vào. Khi bạn triệu gọi hàm **Read()** thì một công tác đọc dữ liệu cho đầy buffer từ đĩa được tiến hành. Tuy nhiên, để cho có hiệu năng, hệ điều hành thường phải đọc trong một lúc một khối lượng lớn dữ liệu tạm thời trữ trên buffer. Buffer hoạt động như một kho hàng.

Một đối tượng *buffered stream* cho phép hệ điều hành tạo buffer riêng cho mình dùng, rồi đọc dữ liệu vào hoặc viết dữ liệu lên ổ đĩa theo một khối lượng dữ liệu nào đó mà hệ điều hành thấy là có hiệu năng. Tuy nhiên, bạn cũng có thể ấn định chiều dài khối dữ liệu. Nhưng bạn nhớ cho là buffer sẽ chiếm chỗ trong ký ức chứ không phải trên đĩa từ. Hiệu quả sử dụng đến buffer là việc xuất nhập dữ liệu chạy nhanh hơn.

Một đối tượng **BufferedStream** được hình thành xung quanh một đối tượng **Stream** mà bạn đã tạo ra trước đó. Muốn sử dụng đến một **BufferedStream** bạn bắt đầu tạo một đối tượng **Stream** thông thường như trong thí dụ 1-8:

```
Stream inputStream = File.OpenRead(@"C:\test\source\folder3.cs");  
Stream outputStream = File.OpenWrite(@"C:\test\source\folder3.bak");
```

Một khi bạn đã có stream bình thường, bạn trao đối tượng này cho hàm constructor của buffered stream:

```
BufferedStream bufInput = new BufferedStream(inputStream);  
BufferedStream bufOutput = new BufferedStream(outputStream);
```

Sau đó, bạn sử dụng **BufferedStream** như là một stream bình thường, bạn triệu gọi hàm **Read()** hoặc **Write()** như bạn đã làm trước kia. Hệ điều hành lo việc quản lý vùng đệm:

```
while ((bytesRead = bufInput.Read(buffer, 0, SIZE_BUFF)) > 0)  
{  
    bufOutput.Write(buffer, 0, bytesRead);  
}
```

Chỉ có một khác biệt mà bạn phải nhớ cho là phải tuân ghi (flush) nội dung của buffer khi bạn muốn bảo đảm là dữ liệu được ghi lên đĩa.

```
bufOutput.Flush();
```

Lệnh trên bảo hệ điều hành lấy toàn bộ dữ liệu trên buffer cho tuân ra ghi lên tập tin trên đĩa. Thí dụ 1-11 liệt kê toàn bộ chương trình:

**Thí dụ 1-11: Sử dụng *BufferedStream* như thế nào?**

```

*****
namespace Progr_CSharp
{
    using System;
    using System.IO;

    class Tester
    {
        const int SizeBuff = 1024;

        public static void Main()
        {
            Tester t = new Tester();
            t.Run();
        }

        // Cho chạy với tên thư mục
        private void Run()
        {
            // tập tin đọc vào và viết ra
            Stream inputStream = File.OpenRead(
                @"C:\test\source\folder3.cs");
            Stream outputStream = File.OpenWrite(
                @"C:\test\source\folder3.bak");
            // thêm buffered stream trên đầu binary stream
            BufferedStream bufInput = new BufferedStream(inputStream);
            BufferedStream bufOutput = new BufferedStream(outputStream);
            // tạo một buffer và biến cho biết số byte
            byte[] buffer = new byte[SizeBuffer];
            int bytesRead;

            // vừa đọc vừa viết
            while ((bytesRead = bufInput.Read(buffer, 0, SizeBuff)) > 0)
            {
                bufOutput.Write(buffer, 0, bytesRead);
            }
            bufOutput.Flush();
            // dọn dẹp trước khi ra về
            bufInput.Close();
            bufOutput.Close();
        }
    }
}
*****

```

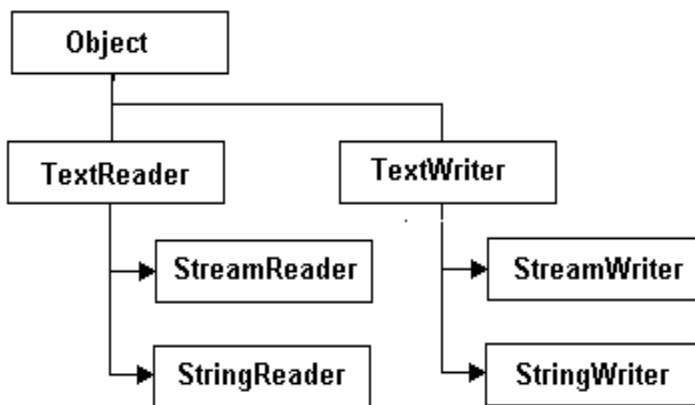
**1.2.5 Làm việc với những tập tin văn bản**

Nếu bạn biết tập tin bạn đang làm việc (đọc/viết) thuộc loại văn bản, nghĩa là dữ liệu dựa trên ký tự (kiểu string), thì bạn nên nghĩ đến việc sử dụng đến các lớp **StreamReader** và **StreamWriter**. Cả hai lớp, theo mặc nhiên, làm việc với ký tự Unicode. Tuy nhiên, bạn có thể thay đổi điều này bằng cách cung cấp một đối tượng qui

chiều được cấu hình một cách thích hợp theo **System.Text.Reference**. Nói tóm lại hai lớp này được thiết kế để thao tác dễ dàng các tập tin loại văn bản.

Lớp **StreamReader** được dẫn xuất từ một lớp abstract mang tên **TextReader**, cũng giống như lớp **StringReader**. Bạn xem hình 1-12. Lớp cơ bản **TextReader** cung cấp một số chức năng hạn chế cho mỗi hậu duệ, đặc biệt khả năng đọc và “liếc nhìn” (peek) lên một dòng ký tự (character stream).

Lớp **StreamWriter** (và **StringWriter**) cũng được dẫn xuất từ một lớp abstract mang



Hình 1-12: Reader và Writer

tên **TextWriter**; lớp này định nghĩa những thành viên cho phép các lớp dẫn xuất viết những dữ liệu văn bản lên một dòng văn bản nào đó. Hình 1-12 cho thấy các mối liên hệ giữa các lớp vừa kể trên:

Muốn biết khả năng viết của lớp **StreamWriter**, bạn cần xem xét chức năng cơ bản được kế thừa từ lớp **TextWriter**. Lớp cơ bản này

định nghĩa một số thành viên theo bảng 1-11 sau đây:

Bảng 1-11: Các thành viên của lớp **TextWriter**

Tên thành viên	Ý nghĩa
<b>Close()</b>	Cho đóng lại writer và giải phóng mọi nguồn lực chiếm dụng. Trong tiến trình này, hệ điều hành sẽ cho vét tuôn ghi (flush) những dữ liệu còn sót lại trong buffer.
<b>Flush()</b>	Hàm này cho xóa sạch tất cả các buffer đối với writer hiện hành và cho ghi lên đĩa tất cả dữ liệu còn sót lại trên buffer, nhưng lại không đóng lại writer.
<b>NewLine</b>	Thuộc tính này dùng làm “hằng sang hàng” (new line constant). Trị mặc nhiên là cặp ký tự CR/LF (“\r\n”).
<b>Write()</b>	Viết một hàng lên text stream không có newline constant.
<b>WriteLine()</b>	Viết một hàng lên text stream với một newline constant.

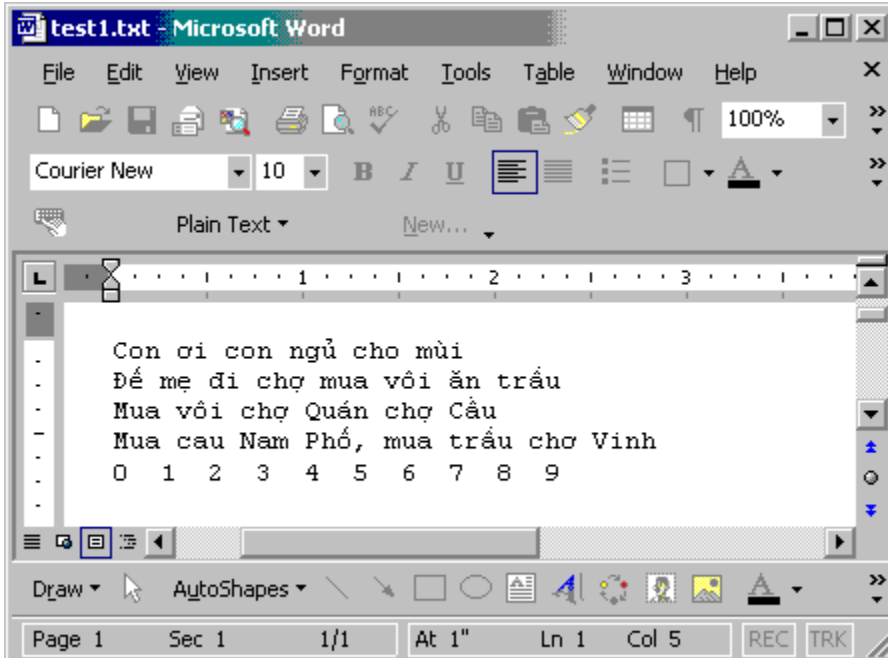
Chắc bạn đã quá quen thuộc với hai hàm chốt **Write()** và **WriteLine()**. Nếu bạn còn nhớ, lớp **System.Console** cũng có những hàm thành viên tương tự viết dữ liệu văn bản lên thiết bị kết xuất chuẩn. Còn ở đây, **TextWriter** thì viết dữ liệu văn bản lên một tập tin được khai báo.

Lớp dẫn xuất **StreamWriter** cung cấp thi công thích ứng đối với các hàm hành sự **Write()**, **Close()** và **Flush()**, cũng như định nghĩa thuộc tính bổ sung **AutoFlush()**. Thuộc tính này, khi cho về true, sẽ ép **StreamWriter** cho flush tất cả dữ liệu mỗi lần bạn thi hành một tác vụ viết. Bạn để ý là nếu cho thuộc tính **AutoFlush** về false, thì hiệu quả hơn, với điều kiện là bao giờ bạn cũng phải triệu gọi hàm **Close()** khi bạn làm việc với **StreamWriter**.

### 1.2.5.1 Viết và Đọc một tập tin văn bản

#### 1.2.5.1.1 Viết một tập tin văn bản đơn giản

Bây giờ, ta thử xem một thí dụ sử dụng lớp **StreamWriter**. Ta tạo một lớp để tạo một tập tin **test1.txt** sử dụng đến lớp **FileInfo**. Sử dụng đến hàm **CreateText()**, bạn có thể có một đối tượng **StreamWriter** hợp lệ. Sau đó, bạn thêm vài hàng dữ liệu văn bản lên tập tin **test1.txt** theo thí dụ 1-12 như sau, và hình 1-13 cho thấy kết quả tập tin **test1.txt** in ra trên Word:



Hình 1-13: Viết một tập tin văn bản, nội dung test1.txt

**Thí dụ 1-12(a): Viết một tập tin văn bản.**

```

public class MyStreamWriterReader
{
    public static int Main(string[] args)
    {
        // Tạo một tập tin
        FileInfo f = new FileInfo(@"D:\Test\test1.txt");

        // Đi lấy một đối tượng StreamWriter rồi viết cái gì đó
        StreamWriter writer = f.CreateText();
        writer.WriteLine("Con ơi con ngủ cho mùi");
        writer.WriteLine("Đề mẹ đi chợ mua vôi ăn trầu");
        writer.WriteLine("Mua vôi chợ Quán chợ Cầu");
        writer.WriteLine("Mua cau Nam Phổ, mua trầu chợ Vinh");
        // Viết ra 10 con số
        for (int i = 0; i < 10; i++)
        {
            writer.Write(i + " ");
        }
        writer.Write(writer.NewLine); // chèn một CR
        // dọn dẹp
        writer.Close();
        Console.WriteLine("Tạo một tập tin hát ru...");
    }
}
*****

```

**1.2.5.1.2 Đọc một tập tin văn bản đơn giản**

Bây giờ bạn cần biết làm thế nào thông qua chương trình đọc dữ liệu từ một tập tin sử dụng đến lớp **StreamReader** tương ứng. Như bạn đã biết lớp này được dẫn xuất từ lớp **TextReader**. Lớp này cung cấp những chức năng được liệt kê theo bảng 1-12 sau đây:

**Bảng 1-12: Các thành viên của lớp TextReader**

Tên thành viên	Ý nghĩa
<b>Peek()</b>	Hàm này trả về ký tự kế tiếp có sẵn mà không phải thay đổi vị trí của reader. Peek có nghĩa là nhìn liếc xéo.
<b>Read()</b>	Đọc dữ liệu từ một input stream.
<b>ReadBlock()</b>	Hàm này đọc một số tối đa ký tự đếm được từ stream hiện hành rồi cho viết dữ liệu lên một buffer, bắt đầu từ chỉ mục.
<b>ReadLine()</b>	Đọc một hàng ký tự từ stream hiện hành và trả về dữ liệu như là một chuỗi. (Một chuỗi null cho biết là EOF – End Of File).
<b>ReadToEnd()</b>	Đọc tất cả các ký tự từ vị trí hiện hành cho tới cuối <b>Text Reader</b> và trả về như là một chuỗi.

Bây giờ, nếu bạn nói rộng lớp `MyStreamWriterReader` để dùng `StreamReader`, bạn có thể đọc dữ liệu văn bản trên tập tin `test1.txt` mà bạn đã viết ra trước đó. Sau đây là phần bổ sung:

***Thí dụ 1-12(b): Viết một tập tin văn bản, rồi đọc lại.***

```
*****
public class MyStreamWriterReader
{
    public static int Main(string[] args)
    {    // Giống như phần trước thí dụ 1-12(a)

        // Bây giờ đọc lại sử dụng đến StreamReader
        Console.WriteLine("Coi xem bạn viết gì: \n");
        StreamReader sr = File.OpenText(@"D:\Test\Test1.txt");
        string input = null;
        while ((input = sr.ReadLine()) != null)
        {    Console.WriteLine(input);
        }
        sr.Close();
        return 0;
    }
}
*****
```

Kết quả giống như trên hình 1-13.

Trong thí dụ trên, hàm hành sự static **File.OpenText()** đem lại cho bạn một đối tượng **StreamReader** hợp lệ. Muốn đọc toàn bộ nội dung của tập tin, bạn nên dùng hàm **ReadToEnd()** như sau:

```
// Tôi muốn đọc trọn
string allOfTheData = sr.ReadToEnd();
MessageBox.Show(allOfTheData, "Here it is:");
sr.Close();
```

### ***1.2.5.1.3 Đọc và viết một tập tin văn bản cùng lúc***

Sau đây là một thí dụ đọc một tập tin rồi viết qua một tập tin khác, cũng sử dụng đến hai lớp **StreamReader** và **StreamWriter**.

Muốn tạo một thể hiện lớp **StreamReader**, bạn bắt đầu tạo một đối tượng **FileInfo** rồi sau đó gọi hàm **OpenText()** đối với đối tượng này:

```
FileInfo theSourceFile = new FileInfo(@"D:\Test\test1.cs");
StreamReader reader = theSourceFile.OpenText();
```



**OpenText()** trả về một đối tượng **StreamReader** đối với tập tin theSourceFile. Với đối tượng stream có trong tay, bây giờ bạn có thể đọc từng hàng văn bản một cho đến EOF:

```
do
{   text = reader.ReadLine();
} while (text != null);
```

**ReadLine()** sẽ đọc vào một lúc một hàng cho tới khi đến cuối tập tin (EOF). Khi đến cuối tập tin thì **StreamReader** sẽ trả về **null**.

Muốn tạo một đối tượng **StreamWriter** bạn cho triệu gọi hàm constructor của **StreamWriter** trao cho hàm này trọn tên tập tin mà bạn muốn viết lên.

```
StreamWriter writer = new StreamWriter(@"D:\Test\test.bak, false);
```

Thông số thứ hai trên hàm này là đối mục bool **append**. Nếu tập tin đã hiện hữu, **true** sẽ cho ghi nối đuôi vào cuối tập tin, và **false** sẽ làm cho dữ liệu cũ bị viết đè chồng lên. Trong trường hợp của chúng ta trao qua **false** sẽ viết chồng lên nếu tập tin hiện hữu.

Bây giờ bạn có thể tạo một vòng lặp viết ra nội dung mỗi hàng của tập tin cũ lên tập tin mới, và cho in ra trên console:

```
do
{   text = reader.ReadLine();
    writer.WriteLine(text);
    Console.WriteLine(text);
} while (text != null);
```

Thí dụ 1-13 liệt kê toàn bộ chương trình

***Thí dụ 1-13: Đọc một tập tin văn bản và viết lên một tập tin văn bản khác.***

\*\*\*\*\*

```
namespace Progr_CSharp
{
    using System;
    using System.IO;

    class Tester
    {
        public static void Main()
        {   Tester t = new Tester();
            t.Run();
        }
        // Cho chạy với tên thư mục
        private void Run()
        {   // cho mở một tập tin và tạo một text reader
            // đối với tập tin này
            FileInfo theSourceFile = new FileInfo(@"D:\Test\test1.cs");
            StreamReader reader = theSourceFile.OpenText();
```

```
// tạo một text writer đối với tập tin mới này
StreamWriter writer = new StreamWriter(
    @"D:\Test\test.bak, false);

string text; // biến dùng trữ mỗi hàng

// rào qua tập tin và đọc vào mỗi hàng
// và viết lên tập tin cũng như lên console
do
{
    text = reader.ReadLine();
    writer.WriteLine(text);
    Console.WriteLine(text);
} while (text != null);
// dọn dẹp trước khi ra về
reader.Close();
writer.Close();
}
}
}
*****
```

Khi chương trình này chạy, nội dung của tập tin nguyên thủy sẽ được viết lên một tập tin mới đồng thời lên màn hình. Bạn để ý cú pháp viết lên console:

```
Console.WriteLine(text);
```

gần giống như cú pháp viết lên tập tin:

```
writer.WriteLine(text);
```

Khác biệt chủ yếu là hàm hành sự **WriteLine()** của **Console** là static, trong khi **WriteLine()** của **StreamWriter**, được kế thừa từ **TextWriter** là một hàm hành sự thể hiện và do đó phải được triệu gọi lên một đối tượng thay vì trên bản thân lớp.

## 1.2.6 Làm việc với StringWriter và StringReader

Khi sử dụng đến hai lớp **StringWriter** và **StringReader**, bạn có thể xử lý thông tin văn bản như là một dòng ký tự trong ký ức (in-memory character). Điều này có thể hữu ích khi bạn muốn gắn nối đuôi thông tin kiểu ký tự lên một buffer nằm sau. Muốn truy xuất vào buffer từ một đối tượng **StringWriter**, bạn có thể triệu gọi hàm **ToString()** (để nhận một đối tượng lớp **System.String**) hoặc hàm hành sự **GetStringBuilder()**, trả về một đối tượng **StringBuilder**. Bạn còn nhớ, ở chương 11, Tập I, “Chuỗi chữ và biểu thức regular”, lớp **System.Text.StringBuilder** cho phép bạn trực tiếp thay đổi một buffer kiểu chuỗi.

Để minh họa, ta thử viết lại thí dụ MyStreamWriterReader, 1-12(a), để viết thông tin kiểu ký tự lên một đối tượng **StringWriter** thay vì trên một tập tin được kết sinh. Bạn có thể thấy là chương trình 1-12(a) và 1-14 dưới đây hầu như giống nhau, vì cả hai lớp **StringWriter** và **StreamWriter** đều kế thừa cùng chức năng của lớp cơ bản **TextWriter**: Hình 1-14 cho thấy kết xuất:

***Thí dụ 1-14: Viết một tập tin văn bản sử dụng lớp StringWriter.***

```
*****
public class MyStringWriterReader
{
    public static int Main(string[] args)
    { // Đi lấy một đối tượng StringWriter rồi viết cái gì đó
        StringWriter writer = new StringWriter();
        writer.WriteLine("Don't forget Mother's Day this year...");
        writer.WriteLine("Don't forget Mother's Day this year...");
        writer.WriteLine("Don't forget these numbers");
        for (int i = 0; i < 10; i++)
        { writer.Write(i + " ");
        }
        writer.Write(writer.NewLine); // chèn một CR

        // dọn dẹp
        writer.Close();
        Console.WriteLine("Stored thoughts in StringWriter...");

        // Lấy một bản sao nội dung (được trữ trong một string) và
        // bơm nó ra console
        Console.WriteLine("Contents: {0}", writer.ToString());
        return 0;
    }
}
*****
```

Bạn cho thử chương trình này thì kết quả giống như hình 1-14 dưới đây phân trên.

```

D:\Thien\SACH_C#\CH_15\ConsoleApplication7\bin\Debug\Console...
Stored thoughts in a StringWriter...
Contents: Don't forget Mother's Day this year...
Don't forget Mother's Day this year...
Don't forget these numbers
0 1 2 3 4 5 6 7 8 9

StringBuilder says:
Don't forget Mother's Day this year...
Don't forget Mother's Day this year...
Don't forget these numbers
0 1 2 3 4 5 6 7 8 9

New StringBuilder says:
Don't forget Mother's INSERTED STUFFs Day this year...
Don't forget Mother's Day this year...
Don't forget these numbers
0 1 2 3 4 5 6 7 8 9

Original says:
Don't forget Mother's Day this year...
Don't forget Mother's Day this year...
Don't forget these numbers
0 1 2 3 4 5 6 7 8 9

```

Hình 01-14: Làm việc với StringBuilder

Bây giờ bạn thêm phần sau đây:

```

// dành cho lớp StringBuilder
using System.Text;

public class MyStringWriterReader
{
    public static int Main(string[] args)
    {
        // Đi lấy một đối tượng StreamWriter rồi viết cái gì đó
        ... giống như trước
        // Đi lấy StringBuilder nội tại
        StringBuilder str = writer.GetStringBuilder();
        string allOfTheData = str.ToString();
        Console.WriteLine("StringBuilder says:\n{0} ", allOfTheData);

        // Chèn item vào buffer ở vị trí 20
        str.Insert(20, "INSERTED STUFF");
        allOfTheData = str.ToString();
        Console.WriteLine("New StringBuilder says:\n{0} ", allOfTheData);

        // gỡ bỏ chuỗi chèn vào
        str.Remove(20, "INSERTED STUFF".Length);
        allOfTheData = str.ToString();
        Console.WriteLine("Original says:\n{0} ", allOfTheData);
        return 0;
    }
}

```

```
}
*****
```

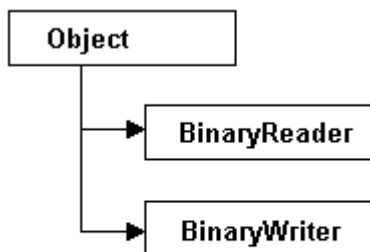
Trong phần này bạn viết vài dòng dữ liệu kiểu ký tự vào đối tượng **StringWriter**, rồi trích và thao tác trên một bản sao nội dung sử dụng đến các hàm thành viên của lớp **StringBuilder**. Hình 1-14 cho thấy kết quả:

Lớp **StringReader** hoạt động cũng tương tự như lớp **StreamReader**. Thật ra, lớp **StringReader** chỉ làm mỗi một việc là phủ quyết các thành viên kế thừa để đọc một khối dữ liệu thay vì một tập tin, như ta thấy sau đây:

```
// Bây giờ tuôn xổ ra (dump) dùng một StringReader
StringReader sr = new StringReader(writer.ToString());
string input = null;
while ((input = sr.ReadLine()) != null)
{
    Console.WriteLine(input);
}
sr.Close();
```

Nếu bạn quan sát kỹ các ứng dụng trước, có thể bạn nhận ra rằng một hạn chế của các hậu duệ của các lớp **TextReader** và **TextWriter**. Không có lớp nào có thể có khả năng cho phép thi hành trực truy (direct access) nội dung dữ liệu. Nghĩa là không có một hàm thành viên nào cho phép bạn đặt để lại con trỏ nội tại (internal cursor) của tập tin hoặc nhảy về một số ký tự rồi mới bắt đầu đọc. Muốn có những chức năng này bạn cần sử dụng đến những hậu duệ khác nhau từ lớp **Stream**.

## 1.2.7 Làm việc với dữ liệu nhị phân (các lớp **BinaryReader** & **BinaryWriter**)



Hình 01-15: **BinaryReader** và **BinaryWriter**

Hai lớp cốt lõi cuối cùng của namespace **System.IO** là **BinaryReader** và **BinaryWriter**, cả hai được dẫn xuất trực tiếp từ **Object**, như theo hình 1-15.

Hai lớp này cho phép bạn đọc và viết kiểu dữ liệu rời rạc (discrete) qua một dòng dữ liệu nằm sau. Lớp **BinaryWriter** định nghĩa một hàm hành sự nạp chồng khá nặng mang tên **Write()** để đưa kiểu dữ liệu vào stream tương ứng. Còn lớp **BinaryReader** cũng cung cấp một vài thành viên

khá quen thuộc. Xem bảng 1-13.

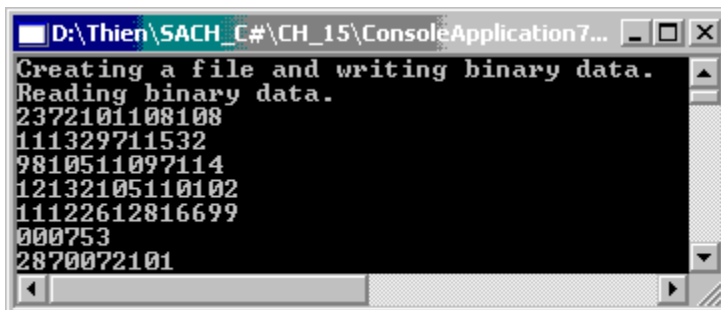
**Bảng 1-13: Các thành viên cốt lõi của lớp *BinaryWriter***

Tên thành viên	Ý nghĩa
<b>BaseStream</b>	Tượng trưng cho stream nằm ẩn sau dùng với binary reader.
<b>Close()</b>	Cho đóng lại binary stream.
<b>Flush()</b>	Hàm này vét tuôn binary stream.
<b>Seek()</b>	Cho đặt để vị trí con trỏ trên stream hiện hành.
<b>Write()</b>	Viết một trị lên stream hiện hành.

Lớp **BinaryReader** bổ sung chức năng cung cấp bởi lớp **BinaryWriter** với những thành viên được liệt kê dưới đây, bảng 1-14:

**Bảng 1-14: Các thành viên cốt lõi của lớp *BinaryReader***

Tên thành viên	Ý nghĩa
<b>BaseStream</b>	Cho hiệu lực việc truy xuất vào stream nằm ẩn.
<b>Close()</b>	Cho đóng lại binary reader
<b>PeekChar()</b>	Hàm này trả về ký tự kế tiếp có sẵn nhưng không dịch tới vị trí trên stream.
<b>Read()</b>	Cho đọc một loạt byte hoặc ký tự và cho trữ trên một bản dãy được khai báo.
<b>ReadXXXX()</b>	Lớp BinaryReader định nghĩa một số hàm ReadXXXX khác nhau lo tùm kiểu dữ liệu kế tiếp từ stream (ReadBoolean(), ReadByte(), ReadInt32(), v.v).

**Hình 01-16: Đọc / Viết nhị phân.**

BinaryWriter. Bạn nhớ cho là hàm constructor của BinaryWriter lấy bất cứ đối tượng nào được dẫn xuất từ Stream (FileStream, MemoryStream hoặc BufferedStream). Một khi dữ liệu đã được viết xong, một đối tượng BinaryReader tương ứng có thể đọc từng byte một, như theo thí dụ 1-15 sau đây: hình 1-16 là kết xuất.

Lớp tự tạo **ByteTweaker** dưới đây viết ra một số kiểu dữ liệu ký tự lên một tập tin \*.dat mới được tạo và được mở, sử dụng đến lớp **FileStream**. Một khi bạn đã có một đối tượng FileStream hợp lệ, bạn trao đối tượng này cho hàm constructor của lớp

**Thí dụ 1-15: Viết rồi đọc một tập tin nhị phân sử dụng lớp *BinaryWriter* và *BinaryReader*.**

```
*****
public class ByteTweaker
{
    public static int Main(string[] args)
    {
        Console.WriteLine("Creating a file and writing binary data...");
        FileStream myFStream = new FileStream(@"D:\Test\temp.dat",
        FileMode.OpenOrCreate, FileAccess.ReadWrite);
        // Viết vài thông tin binary
        BinaryWriter binWriter = new BinaryWriter(myFStream);
        binWriter.Write("Hello as binary info...");
        int myInt = 99;
        float myFloat = 9984.82343F;
        bool myBool = false;
        char[] myCharArray = { 'H', 'e', 'l', 'l', 'o' };
        binWriter.Write(myInt);
        binWriter.Write(myFloat);
        binWriter.Write(myBool);
        binWriter.Write(myCharArray);
        // Reset vị trí nội tại
        binWriter.BaseStream.Position = 0;
        // Đọc thông tin binary như là byte thô
        Console.WriteLine("Reading binary data...");
        BinaryReader binRead = new BinaryReader(myFStream);
        int temp = 0;
        while (binRead.PeekChar() != -1)
        {
            Console.Write(binRead.ReadByte());
            temp = temp + 1;
            if (temp == 5)
            {
                temp = 0;
                Console.WriteLine();
            }
        }
        // Dọn dẹp
        binWriter.Close();
        binRead.Close();
        myFStream.Close();
        return 0;
    }
}
*****
```

## 1.3 Xuất nhập dữ liệu bất đồng bộ (Asynchronous I/O)

Tất cả các chương trình mà bạn đã nghiên cứu trong thời gian qua hoạt động theo kiểu *xuất nhập đồng bộ* (synchronous I/O), nghĩa là trong chương trình bạn đang đọc hoặc viết thì mọi hoạt động khác đều ngưng lại. Phải nói là đọc dữ liệu hoặc viết dữ liệu đòi hỏi thời gian (tương đối) khá dài, đặc biệt khi chờ trễ lâu dài là đĩa từ hoặc mạng.

Với những tập tin đồ sộ, hoặc khi đọc/viết thông qua mạng, thì bạn nên chọn hoạt động theo kiểu *bất đồng bộ xuất nhập* (asynchronous I/O), cho phép bạn bắt đầu cho đọc rồi quay qua làm chuyện gì khác trong khi CLR lo hoàn thành yêu cầu đọc của bạn. .NET Framework cung cấp asynchronous I/O thông qua các hàm hành sự **BeginRead()** và **BeginWrite()** thuộc lớp **Stream**.

Thứ tự hoạt động như sau: bạn cho triệu gọi hàm **BeginRead()** đối với tập tin của bạn, rồi quay qua làm việc gì đó không dính dáng cho với việc đọc tập tin kể trên, trong khi việc đọc tiếp tục tiến hành trên một mạch trình (thread) khác. Khi việc đọc hoàn tất, bạn sẽ được thông báo bởi một hàm hành sự callback. Lúc này bạn có thể xử lý dữ liệu vừa mới được đọc vào, rồi khởi động lại một việc đọc khác, rồi trở qua một việc khác.

Ngoài 3 thông số mà bạn dùng trong việc đọc nhị phân (buffer, di số, và bao nhiêu byte phải đọc) **BeginRead()** còn đòi hỏi một *delegate* và một *tình trạng đối tượng* (state object).

Delegate là một hàm hành sự callback tùy chọn, nếu được cung cấp sẽ được triệu gọi hàm khi dữ liệu được đọc vào xong. Tình trạng đối tượng cũng là tùy chọn. Trong thí dụ này, ta trao **null** đối với tình trạng đối tượng. Tình trạng đối tượng được giữ trong những biến thành viên của lớp trắc nghiệm.

Bạn tự do đưa bất cứ đối tượng nào bạn muốn vào thông số tình trạng và bạn có thể tìm lại nó khi bạn được gọi lại. Điển hình là bạn cất giấu các trị tình trạng mà bạn sẽ cần tìm trở lại. Thông số tình trạng có thể được sử dụng bởi lập trình viên để cầm giữ tình trạng triệu gọi (paused, pending, running, v.v.).

Trong thí dụ này, bạn sẽ tạo một buffer và đối tượng **Stream** như là một biến thành viên private thuộc lớp:

```
public class AsyncIOTester
{
    private Stream inputStream;
    private byte[] buffer;
    const int BufferSize = 256;
    ...
}
```



Ngoài ra, bạn tạo delegate như là thành viên private của lớp:

```
private AsyncCallback myCallBack;    // delegated method
```

Delegate được khai báo thuộc kiểu **AsyncCallback**, chính là điều mà hàm hành sự **Stream.BeginRead()** chờ đợi. Một delegate **AsyncCallback** được khai báo trong namespace **System** như sau:

```
public delegate void AsyncCallback(IAsyncResult ar);
```

Do đó, delegate này có thể được gắn kết với bất cứ hàm hành sự nào trả về void và lấy một thông số là giao diện **IAsyncResult**. CLR sẽ trao vào lúc chạy đối tượng giao diện **IAsyncResult** khi hàm hành sự được triệu gọi. Bạn chỉ cần khai báo hàm hành sự:

```
void OnCompletedRead(IAsyncResult asyncResult)
```

rồi móc delegate vào hàm constructor:

```
public AsynchIOTester()  
{    // ...  
    myCallBack = new AsyncCallback(this.OnCompletedRead);  
    ...  
}
```

Sau đây là cách làm theo từng bước một. Trên Main(), bạn tạo một thể hiện lớp và cho chạy:

```
public static void Main()  
{    AsynchIOTester theApp = new AsynchIOTester();  
    theApp.Run();  
}
```

Khi gọi **new** thì hàm constructor được gọi vào. Trong hàm constructor, bạn cho mở một tập tin và lấy trở lại đối tượng **Stream**. Bạn cấp phát ký ức cho buffer và móc cơ chế callback:

```
public AsynchIOTester()  
{    inputStream = File.OpenRead(@"D:\Test\TestText.txt");  
    buffer = new byte[BufferSize];  
    myCallBack = new AsyncCallback(this.OnCompleteRead);  
}
```

**Bạn để ý** Thí dụ này cần một tập tin văn bản TestText đồ sộ. Do đó, bạn cần bất cứ tập tin văn bản nào và nằm ở thư mục nào cũng được.

Trong hàm hành sự **Run()**, bạn triệu gọi hàm **BeginRead()**, cho phép đọc bất đồng bộ tập tin:

```
void Run()
{   inputStream.BeginRead(buffer,        // chỗ trữ dữ liệu được đọc vào
                                0,         // di số
                                buffer.Length, // kích thước buffer
                                myCallBack,  // call back delegate
                                null);      // tình trạng đối tượng cục bộ
}
```

Sau đó, bạn đi làm chuyện khác. Trong trường hợp này bạn mô phỏng một việc hữu ích, chẳng hạn “đếm con cừu” đến 500.000 lần (giống như lúc bạn mất ngủ), cho hiển thị tiến trình cứ 1000 lần:

```
for (long i = 0; i < 500000; i++)
{   if(i % 1000 == 0)
    {   Console.WriteLine("i: {0}", i);
    }
}
```

Khi tác vụ đọc hoàn tất, CLR sẽ gọi hàm callback của bạn:

```
void OnCompletedRead(IAsyncResult asyncResult)
{
    ...
}
```

Điều đầu tiên khi được thông báo là việc đọc dữ liệu đã hoàn tất là tìm xem bao nhiêu bytes hiện được đọc vào. Muốn thế, bạn triệu gọi hàm **EndRead()** của đối tượng **Stream**, trao cho hàm đối tượng giao diện **IAsyncResult** được trao qua bởi CLR:

```
int bytesRead = inputStream.EndRead(asyncResult);
```

Hàm **EndRead()** trả về số byte đọc được. Nếu số này dương, bạn chuyển đổi buffer thành một chuỗi rồi cho viết lên console, rồi sau đó, triệu gọi hàm **BeginRead()** lần nữa để tiến hành một tác vụ đọc bất đồng bộ.

```
if (bytesRead > 0)
{   String s;
    Encoding.ASCII.GetString (buffer, 0, bytesRead);
    Console.WriteLine(s);
    inputStream.BeginRead(buffer, 0, buffer.Length, myCallBack, null);
}
```

Kết quả là bạn có thể làm việc khác trong khi tác vụ đọc tiến hành, nhưng bạn có thể xử lý dữ liệu mỗi lần buffer đầy tràn dữ liệu. Thí dụ 1-16 liệt kê toàn bộ chương trình:

***Thí dụ 1-16: Viết một chương trình asynchronous I/O***

```
*****
namespace Prog_CSharp
(
    using System;
    using System.IO;
    using System.Threading;
    using System.Text;

    public class AsyncIOTester
    { private Stream inputStream;
      private byte[] buffer;           // buffer chứa dữ liệu đọc vào
      const int BufferSize = 256;      // kích thước buffer
      private AsyncCallback myCallBack; // delegated method
      // Hàm constructor
      public AsyncIOTester()
      { // mở input stream
        inputStream = File.OpenRead(@"D:\Test\TestText.txt");
        buffer = new byte[BufferSize]; // cấp phát buffer
        // gán call back
        myCallBack = new AsyncCallback(this.OnCompleteRead);
      }

      public static void Main()
      { // tạo một thể hiện AsyncIOTester để triệu gọi hàm constructor
        AsyncIOTester theApp = new AsyncIOTester();
        theApp.Run();
      }

      void Run()
      { inputStream.BeginRead(buffer, // chỗ trữ dữ liệu được đọc vào
                              0,      // di số
                              buffer.Length, // kích thước buffer
                              myCallBack, // call back delegate
                              null);    // tình trạng đối tượng cục bộ

        // làm việc gì đó khi CLR đang đọc vào dữ liệu
        for (long i = 0; i < 500000; i++)
        { if(i % 1000 == 0)
          { Console.WriteLine("i: {0}", i);
            }
          }
        }

      // Hàm call back
      void OnCompletedRead(IAsyncResult asyncResult)
      { int bytesRead = inputStream.EndRead(asyncResult);
        // nếu có dữ liệu tạo một string cho hiển thị rồi
        // bắt đầu lại
        if (bytesRead > 0)

```

```

        {
            String s = Encoding.ASCII.GetString (buffer, 0, bytesRead);
            Console.WriteLine(s);
            inputStream.BeginRead(buffer, 0, buffer.Length,
                                  myCallBack, null);
        }
    }
}
}
}
}
*****

```

Bạn thử lấy một chương trình văn bản đồ sộ, cho chạy thử rồi sẽ thấy kết quả. Bạn cho thay lỗi tìm về trong câu lệnh

```
inputStream = File.OpenRead(@"D:\Test\TestText.txt");
```

Kết xuất cho thấy là chương trình chạy cùng lúc trên hai mạch trình (thread). Tác vụ đọc được thực hiện ở hậu trường trong khi mạch trình kia đếm số và in ra từng hằng ngàn một. Khi việc đọc hoàn tất, dữ liệu được in ra trên màn hình rồi sau đó trở lại đếm.

Trong các ứng dụng thực tế, có thể phải xử lý các yêu cầu của người sử dụng hoặc tính toán số liệu trong khi asynchronous I/O thì bạn lo tìm dữ liệu hoặc trữ dữ liệu lên một tập tin hoặc lên một căn cứ dữ liệu.

## 1.4 Xuất nhập dữ liệu trên mạng (Network I/O)

Viết lên một đối tượng nằm ở xa trên Internet cũng không khác chi mấy so với việc viết lên một tập tin trên máy tại chỗ của bạn. Có thể bạn muốn làm điều này nếu chương trình của bạn cần trữ dữ liệu trên một tập tin nằm trên máy mạng hoặc bạn muốn tạo một chương trình cho hiển thị thông tin lên một màn hình được nối về một máy tính trên mạng.

Network I/O được dựa trên việc sử dụng đến stream được tạo ra với socket (cái đế, hoặc cái đuôi đèn<sup>1</sup>). Socket rất hữu dụng đối với những ứng dụng client/server, peer to peer (P2P) và khi thực hiện những thủ tục triệu gọi từ xa.

Một socket là một đối tượng tượng trưng cho những điểm dừng cuối (endpoint) trong thông thương giữa các tiến trình liên lạc với nhau thông qua mạng. Socket có thể làm việc với nhiều nghi thức (protocol), bao gồm **UDP** (User Datagram Protocol) và **TCP/IP** (Transmission Control Protocol/Internet Protocol). Trong mục này chúng tôi sẽ tạo một kết nối (connection) TCP/IP giữa một server với một client. TCP/IP là một nghi thức dựa

---

<sup>1</sup> Người Mỹ có một cách đặt từ khá đẹp chút nào.

trên kết nối (connection-based) đối với liên lạc thông qua mạng. Connection-based có nghĩa là với TCP/IP một khi kết nối được thực hiện xong thì hai process có thể nói chuyện với nhau như là chúng đang được nối bởi một đường dây điện thoại trực tiếp.

**Bạn để ý** Mặc dù TCP/IP được thiết kế nói chuyện xuyên mạng, bạn có thể mô phỏng việc liên lạc xuyên mạng bằng cách cho chạy hai process trên cùng một máy.

Trên một máy tính nào đó, có thể có nhiều ứng dụng nói chuyện với những khách hàng khác nhau cùng lúc (nghĩa là có thể bạn đang chạy một web server và đồng thời một FTP server và đồng thời một chương trình chịu hỗ trợ tính toán). Do đó, mỗi ứng dụng phải có mã nhận diện (ID) khác nhau, để khách hàng có thể cho biết ứng dụng nào mình muốn làm việc. Mã ID này được gọi là một **port** (một bến). Bạn có thể hình dung địa chỉ IP là số điện thoại, còn port là máy nối (extension).

Server sẽ cho hiển lộ một socket và yêu cầu socket theo nghe ngóng kết nối trên một port cụ thể nào đó. Hàm constructor đối với socket có một thông số: một số nguyên tượng trưng cho mã ID port mà socket chịu trách nhiệm theo dõi nghe ngóng.

**Bạn để ý** Các ứng dụng khách hàng thường nối về một địa chỉ IP cụ thể. Thí dụ, địa chỉ của Yahoo là 216.114.108.245. Khách hàng buộc phải nối thêm về một port nào đó. Tất cả các Web browser đều nối về port 80 theo mặc nhiên. Mã số port đi từ 0 đến 65.535.

Một khi socket đã được tạo xong, bạn triệu gọi hàm **Start()** đối với socket, cho socket biết là bắt đầu chấp nhận kết nối mạng. Khi server đã sẵn sàng bắt đầu đáp ứng những cuộc gọi từ khách hàng, bạn triệu gọi hàm **Accept()**. Mạch trình theo đây bạn đã triệu gọi **Accept()** sẽ chặn đứng: chờ đợi buồn bã cạnh điện thoại, nghe ngửi những bàn tay ảo tưởng hy vọng một cú gọi.

Bạn có thể tưởng tượng tạo ra socket đơn giản nhất thế giới. Nó chờ đợi một cách kiên nhẫn khách hàng gọi đến, và khi nhận được cuộc gọi thì nó tương tác với khách hàng này bỏ qua tất cả các khách hàng khác. Một ít khách hàng kế tiếp cần gọi sẽ được kết nối nhưng lại biểu ngời chờ một chút. Trong những khách hàng này ngời nghe nhạc và được bảo rằng cuộc gọi của họ rất quan trọng và sẽ được thụ lý theo thứ tự nhận được, họ sẽ khóa chặt mạch trình riêng của họ. Một khi hàng nối đuôi đã đầy nhóc, các cuộc gọi vào sau sẽ nhận tín hiệu đường dây bị bận. Họ phải ngời chờ cho tới khi socket đơn giản của chúng ta xong việc với khách hàng hiện hành. Nói tóm lại mô hình trên chỉ hiệu lực đối với những server chỉ nhận vài ba cuộc gọi trong một tuần, sẽ không thích ứng nổi với những ứng dụng thế giới thực hiện hành. Phần lớn server phải thực kiên hằng ngàn, đôi khi hằng chục ngàn kết nối một phút.

Muốn thụ lý một khối lượng lớn kết nối, các ứng dụng phải hoạt động theo chế độ asynchronous I/O: chấp nhận cuộc gọi và trả về một socket mới với kết nối cho khách hàng. Socket nguyên thủy sau đó trả về cho việc nghe ngóng, chờ đợi khách hàng mới. Theo thể thức này, ứng dụng của bạn có thể thụ lý nhiều cuộc gọi; mỗi lần một cuộc gọi được chấp nhận thì một socket mới sẽ được tạo ra.

Khách hàng sẽ không biết đến mero vật này khi một socket mới được tạo ra. Đối với khách hàng thì họ đã vào đúng IP address và đúng port yêu cầu. Bạn để ý là socket mới thiết lập một kết nối lâu dài với khách hàng. Điều này khác với UDP (User Datagram Protocol) theo đây nó dùng một nghi thức không kết nối (connectionless protocol). Với TCP/IP, một khi kết nối đã được thực hiện xong khách hàng và server biết nói thể nào với nhau khỏi phải định lại địa chỉ (re-address) mỗi packet.

Bản thân lớp Socket khá đơn giản. Nó biết làm thế nào trở thành một điểm cuối, nhưng lại không biết làm thế nào nhận một cuộc gọi và thiết lập một kết nối TCP/IP. Điều này sẽ được thực hiện bởi lớp **TcpListener**. Lớp **TcpListener** xây dựng trên lớp **Socket** để cung cấp những dịch vụ cao cấp TCP/IP.

## 1.4.1 Tạo một Network Streaming Server

Muốn tạo một network server dùng cho TCP/IP streaming, bạn bắt đầu tạo một đối tượng **TcpListener** để nghe ngóng tại port TCP/IP mà bạn đã chọn. Thí dụ ta chọn port 65000 nào đó:

```
TcpListener tcpListener = new TcpListener(65000);
```

Một khi đối tượng **TcpListener** đã được tạo xong, bạn có thể yêu cầu nó bắt đầu nghe ngóng:

```
tcpListener.Start();
```

Bây giờ một khách hàng yêu cầu kết nối:

```
Socket socketForClient = tcpListener.Accept();
```

Hàm hành sự **Accept()** của lớp Listener trả về một đối tượng Socket tượng trưng cho *giao diện Berkeley socket* và được gắn liền với một điểm cuối cụ thể nào đó. **Accept()** là một hàm hành sự asynchronous sẽ không trả về cho tới khi nó nhận được một yêu cầu kết nối.

Nếu socket được kết nối, bạn sẵn sàng chuyển tập tin cho khách hàng:

```
if (socketForClient.Connected)
{
```

Bạn tạo một đối tượng lớp `NetworkStream`, trao socket cho hàm constructor:

```
NetworkStream networkStream = new NetworkStream(socketForClient);
```

Sau đó, bạn tạo một đối tượng `StreamWriter` như bạn đã làm trước đó, ngoại trừ lần này không phải là một tập tin mà đúng ra là đối tượng `NetworkStream` mà bạn vừa tạo ra:

```
System.IO.StreamWriter streamWriter = new
    System.IO.StreamWriter(networkStream);
```

Khi bạn viết lên stream, thì stream này được gửi về cho khách hàng thông qua mạng. Thí dụ 1-17 cho thấy phần chương trình phía server. Chúng tôi cho lược giản đến mức cần thiết tối thiểu. Trong thực tế sản xuất, bạn phải cho đoạn mã xử lý yêu cầu vào trong một mạch trình và đóng khung trong khối **try** để thụ lý những tình huống liên quan đến mạng.

### ***Thí dụ 1-17: Thiết đặt một streaming network server***

```
*****
using System;
using System.Net.Sockets;

public class NetworkIOServer
{
    public static void Main()
    {
        NetworkIOServer app = new NetworkIOServer();
        app.Run();
    }

    private void Run()
    {
        // tạo một TcpListener mới và bắt đầu nghe ngóng trên port 65000
        TcpListener tcpListener = new TcpListener(65000);
        tcpListener.Start();

        // tiếp tục nghe ngóng cho tới khi bạn gửi một tập tin
        for (;;)
        {
            // nếu khách hàng kết nối, chấp nhận kết nối rồi
            // trả về một socket mới socketForClient
            // trong khi tcpListener tiếp tục nghe ngóng
            Socket socketForClient = tcpListener.Accept();
            if (socketForClient.Connected)
            {
                Console.WriteLine("Client connected");

                // triệu gọi hàm helper chuyển đi tập tin
                SendFileToClient(socketForClient);
                Console.WriteLine("Disconnecting from Client");
                // dọn dẹp mà về nhà
                socketForClient.Close();
            }
        }
    }
}
```

```

        Console.WriteLine("Exiting...");
        break;
    }
}

// hàm hỗ trợ gửi tập tin đi
private void SendFileToClient(Socket socketForClient)
{
    // tạo một network stream và một stream writer
    NetworkStream networkStream = new NetworkStream(socketForClient);
    System.IO.StreamWriter streamWriter = new
        System.IO.StreamWriter(networkStream);

    // tạo một stream reader cho tập tin
    System.IO.StreamReader streamReader = new System.IO.StreamReader(
        @"D:\Test\myTest.txt");
    string theString;
    // rảo qua tập tin và gửi đi từng hàng một
    do
    {
        theString = streamReader.ReadLine();
        if (theString != null)
        {
            Console.WriteLine("Sending {0}", theString);
            streamWriter.WriteLine(theString);
            streamWriter.Flush();
        }
    } while (theString != null);

    // dọn dẹp
    streamReader.Close();
    streamWriter.Close();
    networkStream.Close();
}
}
*****

```

## 1.4.2 Tạo một Streaming Network Client

Phía khách hàng sẽ hiển lộ một đối tượng **TcpClient** tượng trưng cho một kết nối client theo TCP/IP về một “chủ chứa” (host):

```

TcpClient socketForServer;
socketForServer = new TcpClient("localhost", 65000);

```

Với đối tượng **TcpClient** này bạn có thể tạo một **NetworkStream**, và trên stream này bạn có thể tạo một **StreamReader**:

```

NetworkStream networkStream = socketForServer.GetStream();
System.IO.StreamReader streamReader = new
    System.IO.StreamReader(networkStream);

```



Bây giờ bạn đọc vào dữ liệu cho tới khi không còn nữa, cho kết xuất lên console:

```
do
{   outputString = streamReader.ReadLine();
    if (outputString != null)
    {   Console.WriteLine(outputString);
    }
} while (outputString != null)
```

Thí dụ 1-18 là toàn bộ chương trình phía client

### ***Thí dụ 1-18: Thiết đặt một streaming network client***

```
*****
using System;
using System.Net.Sockets;

public class NetworkIOClient
{
    public static void Main()
    {   // tạo một TcpClient nói chuyện với server
        TcpClient socketForServer;
        try
        {   socketForServer = new TcpClient("localhost", 65000);
        }
        catch
        {   Console.WriteLine("Failed to connect to server at {0}: 65000",
                                "localhost");
            return;
        }

        // tạo một network stream và một stream reader
        NetworkStream networkStream = socketForServer.GetStream();
        System.IO.StreamReader streamReader = new
            System.IO.StreamReader(networkStream);

        try
        {   string outputString;
            // đọc dữ liệu từ host và cho hiển thị
            do
            {   outputString = streamReader.ReadLine();
                if (outputString != null)
                {   Console.WriteLine(outputString);
                }
            } while (outputString != null)
        }
        catch
        {   Console.WriteLine("Exception reading from Server");
        }
        networkStream.Close(); // dọn dẹp
    }
}
*****
```

Bạn thử trải nghiệm xem trên máy tính của bạn ra sao.

**Bạn để ý** Nếu bạn trải nghiệm chương trình này trên một máy tính duy nhất, bạn cho chạy client và server trên cửa sổ command khác nhau hoặc instance riêng rẽ của môi trường triển khai. Bạn sẽ khởi động server trước tiên hoặc client thất bại cho biết nó không kết nối được.

## 1.4.3 Thụ lý nhiều kết nối

Như đã nói trên, mỗi khách hàng chiếm trọn “đầu óc” máy server. Điều cần thiết cho một server là nó chấp nhận kết nối rồi chuyển kết nối cho xuất nhập phủ chồng (overlapped I/O), cung cấp giải pháp bất đồng bộ như bạn đã dùng trước đó để đọc một tập tin.

Muốn thế, bạn sẽ tạo một server mới, **AsynchNetworkServer**, nó sẽ nằm lòng trong một lớp mới, **ClientHandler**. Khi **AsynchNetworkServer** tiếp nhận một kết nối client, nó sẽ hiển lộ một đối tượng **ClientHandler** và trao socket cho thể hiện này.

Hàm constructor của **ClientHandler** sẽ tạo một bản sao socket và một buffer và sẽ mở một **NetworkStream** trên socket này. Sau đó, nó sẽ dùng overlapped I/O để bất đồng bộ đọc và viết trên socket này. Để minh họa, nó đơn giản sẽ hồi âm đoạn văn bản mà khách hàng gửi trả về lại cho khách hàng đồng thời lên console.

Muốn tạo asynchronous I/O, lớp **ClientHandler** sẽ định nghĩa hai hàm delegate, **OnReadComplete()** và **OnWriteComplete()**, lo việc overlapped I/O đối với những chuỗi chữ khách hàng gửi đến.

Phần thân của hàm **Run()** đối với server cũng tương tự như trên thí dụ 1-17. Trước tiên, bạn tạo một đối tượng listener rồi cho gọi **Start()**. Sau đó, bạn tạo một vòng lặp vô tận và gọi **AcceptSocket()**. Một khi socket đã được kết nối thay vì thụ lý kết nối, bạn tạo một **ClientHandler** mới và triệu gọi hàm **StartRead()** trên đối tượng này.

Sau đây là toàn bộ đoạn mã đối với server liệt kê bởi thí dụ 1-19:

### ***Thí dụ 1-19: Thiết đặt một asynchronous streaming network server***

```
*****
using System;
using System.Net.Sockets;

public class AsynchNetworkServer
{
    class ClientHandler
    {
        public ClientHandler(Socket socketForClient)
```

```

    {
        socket = socketForClient;
        buffer = new byte[256];
        networkStream = new NetworkStream(socketForClient);
        callbackRead = new AsyncCallback(this.OnReadComplete);
        callbackWrite = new AsyncCallback(this.OnWriteComplete);
    }

    // bắt đầu đọc chuỗi chữ từ client
    public void StartRead()
    {
        networkStream.BeginRead(buffer, 0, buffer.Length,
                                callbackRead, null);
    } // end StartRead

    // khi call back bởi tác vụ đọc, cho hiển thị string
    // và hồi âm về cho khách hàng
    private void OnReadComplete(IAsyncResult ar)
    {
        int bytesRead = networkStream.EndRead(ar);
        if (bytesRead > 0)
        {
            string s = System.Text.Encoding.ASCII.GetString (
                buffer, 0, bytesRead);
            Console.WriteLine("Received {0} bytes from client:
                               {0}", bytesRead, s);
            networkStream.BeginWrite(buffer, 0, bytesRead,
                                     callbackWrite, null);
        }
        else
        {
            Console.WriteLine("Read connection dropped");
            networkStream.Close();
            socket.Close();
            networkStream = null;
            socket = null;
        }
    } // end OnReadComplete

    // sau khi viết chuỗi in ra một thông điệp và kết thúc việc đọc
    private void OnWriteComplete(IAsyncResult ar)
    {
        networkStream.EndWrite(ar);
        Console.WriteLine("Write complete");
        networkStream.BeginRead(buffer, 0, buffer.Length,
                                callbackRead, null);
    } // end OnWriteComplete

    private byte[] buffer;
    private Socket socket;
    private NetworkStream networkStream;
    private AsyncCallback callbackRead;
    private AsyncCallback callbackWrite;
} // end ClientHandler

public static void Main()
{
    AsynchNetworkServer app = new AsynchNetworkServer();
    app.Run();
} // end Main

```

```

private void Run()
{
    // tạo một tcpListener mới và bắt đầu nghe ngóng ở port 65000
    TcpListener tcpListener = new TcpListener(65000);
    tcpListener.Start();

    // tiếp tục nghe ngóng cho tới khi gọi đi tập tin
    for(;;)
    {
        // nếu khách hàng kết nối chấp nhận kết nối
        // và trả về một socket mới mang tên socketForClient,
        // trong khi tcpListener tiếp tục nghe ngóng
        Socket socketForClient = tcpListener.AcceptSocket();
        if(socketForClient.Connected)
        {
            Console.WriteLine("Client connected");
            ClientHandler handler =new ClientHandler(socketForClient);
            handler.StartRead();
        }
    } // end for
} // end Run
}
*****

```

Server khởi động và nghe ngóng port 65000. Nếu một khách hàng kết nối, server sẽ hiển lộ một đối tượng **ClientHandler** lo quản lý xuất nhập dữ liệu với khách hàng trong khi server nghe ngóng khách hàng kế tiếp.

**Bạn để ý** Trong thí dụ này, bạn viết chuỗi nhận được từ khách hàng lên console trên **OnReadComplete** và **OnWriteComplete**, và việc viết lên console có thể khoá chặt mạch trình cho tới khi việc viết hoàn tất. Trong một chương trình thực tế, bạn không muốn nên khoá chặt trên các hàm này vì bạn đang sử dụng pooled thread. Nếu bạn khoá chặt **OnReadComplete** và **OnWriteComplete**, bạn có thể gây ra việc nhiều mạch trình được thêm vào thread pool như vậy sẽ không hiệu quả và làm hỏng hiệu năng cũng như tính tăng qui mô (scalability).

Đoạn mã phía khách hàng khá đơn giản. Khách hàng tạo một **tcpSocket** đối với port mà server sẽ nghe ngóng (65000), và tạo một đối tượng **NetworkStream** cho socket này. Sau đó, nó sẽ viết một thông điệp cho stream này và flush buffer. Phía client sẽ tạo một đối tượng **StreamReader** để đọc stream này và viết ra những gì nhận được lên console. Thí dụ 1-20 cho liệt kê toàn bộ chương trình phía client.

### **Thí dụ 1-20: Thiết đặt một asynchronous streaming network client**

```

*****
using System;
using System.Net.Sockets;
using System.Threading;
using System.Runtime.Serialization.Formatters.Binary;

public class AsynchNetWorkClient

```

```

{
    static public int Main()
    {
        AsynchNetworkClient client = new AsynchNetworkClient();
        return client.Run();
    }

    AsynchNetworkClient() // hàm constructor
    {
        string serverName = "localhost";
        Console.WriteLine("Connecting to {0}", serverName);
        TcpClient tcpSocket = new TcpClient(serverName, 65000);
        streamToServer = tcpSocket.GetStream();
    }

    private int Run();
    {
        string message = "Hello Programming C#";
        Console.WriteLine("Sending {0} to server", message);

        // tạo một StreamWriter và dùng nó để viết message lên server
        System.IO.StreamWriter writer = new
            System.IO.StreamWriter(streamToServer);
        writer.WriteLine(message);
        writer.Flush();

        // Đọc câu trả lời
        System.IO.StreamReader reader = new
            System.IO.StreamReader(streamToServer);
        string strResponse = reader.ReadLine();
        Console.WriteLine("Received: {0}", strResponse);
        streamToServer.Close();
        return 0;
    } // end Run
    private NetworkStream streamToServer;
} // end AsynchNetworkClient
*****

```

Trong thí dụ này, server mạng không khoá chặt trong khi nó thụ lý kết nối với khách hàng, nhưng nó ủy quyền quản lý những kết nối này cho những thể hiện của ClientHandler. Các khách hàng không được gặp phải chờ đợi server thụ lý kết nối của họ.

## 1.4.4 Asynchronous Network File Streaming

Bạn có thể phối hợp tài năng học được liên quan đến việc tập tin bất đồng bộ đọc theo asynchronous network streaming để tạo ra một chương trình phục vụ một tập tin cho khách hàng khi được yêu cầu.

Server của bạn bắt đầu bằng một asynchronous read trên socket, chờ nhận một tên tập tin từ khách hàng. Một khi có được tên tập tin, bạn có thể khởi động một asynchronous read tập tin này trên server. Mỗi khi vùng đệm của tập tin đầy nhóc dữ liệu, thì bạn bắt đầu một asynchronous write về cho khách hàng. Khi asynchronous write trên phía khách hàng chấm dứt, bạn có thể khởi động một asynchronous read khác trên tập tin, như vậy bạn nhảy qua nhảy lại, cho điền buffer từ dữ liệu tập tin rồi viết nội dung buffer lên khách hàng. Khách hàng không làm gì cả ngoài việc đọc stream từ server. Trong thí dụ kế tiếp khách hàng sẽ viết nội dung tập tin lên console, nhưng bạn có thể bắt đầu dễ dàng một asynchronous write lên một tập tin mới trên phía client, như vậy tạo một chương trình sao chép tập tin trên mạng.

Cấu trúc của server giống như theo thí dụ 1-19. Một lần nữa bạn sẽ tạo một lớp **ClientHandler** nhưng lần này bạn thêm một **AsyncCallback** mang tên **myFileCallback** mà bạn sẽ khởi gán trên hàm constructor cùng với những callback đối với việc đọc và viết trên mạng:

```
myFileCallback = new AsyncCallback(this.OnFileCompletedRead);  
callbackRead = new AsyncCallback(this.OnReadComplete);  
callbackWrite = new AsyncCallback(this.OnWriteComplete);
```

Hàm **Run()** của lớp nằm ngoài, bây giờ được mang tên **AsyncNetworkFileServer**, không thay đổi. Một lần nữa, bạn tạo và khởi động lớp **TcpListener** và tạo một vòng lặp bất tận theo đây bạn triệu gọi hàm **AcceptSocket()**, và nếu bạn có một socket bạn cho hiển lộ **ClientHandler** và triệu gọi hàm **StartRead()**. Cũng giống như thí dụ trước, **StartRead()** khởi động một **BeginRead()**, trao qua buffer và delegate về **OnReadComplete**.

Khi việc đọc từ stream trên mạng kết thúc, hàm hành sự delegate **OnReadComplete()** được triệu gọi và tìm lại tên tập tin từ buffer. Nếu văn bản được trả về, **OnReadComplete()** tìm thấy lại một chuỗi từ buffer sử dụng đến hàm hành sự static **System.Text.Encoding.ASCII.GetString()**:

```
if (bytesRead > 0)  
{ // chuyển chuỗi cho tên tập tin  
    string fileName = System.Text.Encoding.ASCII.GetString (  
        buffer, 0, bytesRead);
```

Bây giờ bạn có một tên tập tin, bạn có thể mở một stream lên tập tin và sử dụng đúng các đọc tập tin bất đồng bộ được dùng trong thí dụ 1-17.

```
// mở file input stream  
inputStream = File.OpenRead(fileName);  
  
// bắt đầu đọc tập tin  
inputStream.BeginRead(buffer, 0, buffer.Length,  
    myFileCallback, null);
```

Đoạn mã đọc tập tin có riêng callback được triệu gọi hàm khi input stream đọc đầy một buffer từ tập tin nằm phía server.

**Bạn để ý:** Như đã nói trước kia, thông thường bạn sẽ không hành động gì trên overlapped I/O để có thể khóa chặt mạch trình trong một thời gian dài. Phần triệu gọi hàm mở và đọc tập tin thường giao cho một mạch trình hỗ trợ thay vì cho thực hiện trên **OnReadComplete()**. Chúng tôi cho đơn giản hóa trong thí dụ này để tránh rối rắm khi giải thích vấn đề.

Khi buffer đầy, **OnFileCompletedRead()** được triệu gọi, kiểm tra xem có đọc byte nào hay không, và nếu có, thì bắt đầu một asynchronous write lên mạng:

```
if (bytesRead > 0)
{
    // bắt đầu ghi ra client
    networkStream.BeginWrite(buffer, 0, bytesRead,
        callbackWrite, null);
}
```

Khi network write kết thúc, hàm **OnWriteComplete()** được triệu gọi và nó cho khởi động một tác vụ đọc khác từ tập tin:

```
private void OnWriteComplete(IAsyncResult ar)
{
    networkStream.EndWrite(ar);
    Console.WriteLine("Write complete");
    // bắt đầu đọc thêm từ tập tin
    inputStream.BeginRead(buffer, 0, buffer.Length,
        myFileCallback, null);
} // end OnWriteComplete
```

Chu kỳ bắt đầu lần nữa với một tác vụ đọc khác trên tập tin, và chu kỳ cứ tiếp tục cho tới khi tập tin được đọc trọn hết và được chuyển cho client. Đoạn mã phía client đơn giản viết tên tập tin lên network stream để khởi động việc đọc tập tin:

```
string message = @"D:\Test\Testtxt.txt";
Console.WriteLine("Sending {0} to server", message);

// tạo một streamWriter và dùng nó để viết message lên server
System.IO.StreamWriter writer = new
    System.IO.StreamWriter(streamToServer);
writer.WriteLine(message);
writer.Flush();
```

Đoạn mã client bắt đầu một vòng lặp, đọc từ network stream cho tới khi không còn bytes được gửi đi bởi server. Khi server làm xong việc, thì cho đóng lại network stream. Bạn bắt đầu khởi gán một trị bool, fQuit, cho về false và tạo một buffer dùng dữ liệu do server gửi đến:

```
bool fQuit = false;
// khi còn dữ liệu từ server, tiếp tục đọc
while (!fQuit)
{ // buffer cầm giữ trả lời
    char[] buffer = new char[BufferSize];
```

Bây giờ, bạn đã sẵn sàng tạo một **StreamReader** từ biến thành viên **streamToServer** của lớp **NetworkStream**.

```
System.IO.StreamReader reader = new
    System.IO.StreamReader(streamToServer);
```

Hàm **Read()** cần đến 3 thông số: buffer, offset và kích thước của buffer:

```
int bytesRead = reader.Read(buffer, 0, BufferSize);
```

Bạn kiểm tra xem **Read()** có trả về bytes nào hay không. Nếu không thì coi như xong, cho biến **fQuit** về true để vòng lặp thoát ra chấm dứt.

```
if (bytesRead == 0) // none? quit
    fQuit = true;
```

Nếu có nhận byte, bạn có thể viết lên console hoặc lên một tập tin hoặc làm gì đó theo ý muốn của bạn:

```
else
{ // cho hiển thị một chuỗi
    string theString = new String(buffer);
    Console.WriteLine(theString);
}
```

Một khi thoát khỏi vòng lặp, bạn cho đóng lại đối tượng **NetworkStream**.

```
// dọn dẹp
streamToServer.Close();
```

Sau đây là toàn bộ mã nguồn đối với server, thí dụ 1-21, và đối với client, thí dụ 1-22:

### ***Thí dụ 1-21: Thiết đặt một asynchronous streaming network file server***

```
*****
using System;
using System.Net.Sockets;
using System.Text;
using System.IO;
```



```
// lấy tên tập tin từ client, mở tập tin rồi gửi nội dung
// từ server lên cho client
public class AsyncNetworkFileServer
{
    class ClientHandler
    {
        public ClientHandler(Socket socketForClient) // hàm constructor
        {
            socket = socketForClient; // khởi gán biến thành viên
            buffer = new byte[256];    // buffer trữ dữ liệu
            // tạo network stream
            networkStream = new NetworkStream(socketForClient);
            // đặt để file callback cho đọc tập tin
            myFileCallback = new AsyncCallback(this.OnFileCompletedRead);
            // đặt để callback cho đọc và viết
            callbackRead = new AsyncCallback(this.OnReadComplete);
            callbackWrite = new AsyncCallback(this.OnWriteComplete);
        } // end hàm constructor

        // bắt đầu đọc chuỗi chữ từ client
        public void StartRead()
        {
            // đọc từ mạng, lấy tên tập tin
            networkStream.BeginRead(buffer, 0, buffer.Length,
                                     callbackRead, null);
        } // end StartRead

        // khi call back bởi tác vụ đọc, cho hiển thị string
        // và hỏi âm về cho khách hàng
        private void OnReadComplete(IAsyncResult ar)
        {
            int bytesRead = networkStream.EndRead(ar);
            // nếu bạn nhận một chuỗi
            if (bytesRead > 0)
            {
                // chuyển chuỗi cho tên tập tin
                string fileName = System.Text.Encoding.ASCII.GetString (
                    buffer, 0, bytesRead);

                // nhật tu console
                Console.WriteLine("Opening file {0}", fileName);
                // mở file input stream
                inputStream = File.OpenRead(fileName);
                // bắt đầu đọc tập tin
                inputStream.BeginRead(buffer, 0, buffer.Length,
                    myFileCallback, null);
            }
            else
            {
                Console.WriteLine("Read connection dropped");
                networkStream.Close();
                socket.Close();
                networkStream = null;
                socket = null;
            }
        } // end OnReadComplete

        // khi có một buffer đầy nhóc
        private void OnFileCompletedRead(IAsyncResult asyncResult)
        {
            int bytesRead = inputStream.EndRead(asyncResult);
            // nếu bạn đọc tập tin nào đó
            if (bytesRead > 0)
```

```

        { // bắt đầu ghi ra client
            networkStream.BeginWrite(buffer, 0, bytesRead,
                callbackWrite, null);
        }
    } // end OnFileCompletedRead

    // sau khi viết chuỗi, đọc tiếp từ tập tin
    private void OnWriteComplete(IAsyncResult ar)
    {
        networkStream.EndWrite(ar);
        Console.WriteLine("Write complete");

        // bắt đầu đọc thêm từ tập tin
        inputStream.BeginRead(buffer, 0, buffer.Length,
            myFileCallback, null);
    } // end OnWriteComplete

    private const int BufferSize = 256;
    private byte[] buffer;
    private Socket socket;
    private NetworkStream networkStream;
    private Stream inputStream;
    private AsyncCallback callbackRead;
    private AsyncCallback callbackWrite;
    private AsyncCallback myFileCallback;
} // end ClientHandler

public static void Main()
{
    AsynchNetworkFileServer app = new AsynchNetworkFileServer();
    app.Run();
} // end Main

private void Run()
{
    // tạo một tcpListener mới và bắt đầu nghe ngóng ở port 65000
    TcpListener tcpListener = new TcpListener(65000);
    tcpListener.Start();

    // tiếp tục nghe ngóng cho tới khi gọi đi tập tin
    for(;;)
    {
        // nếu khách hàng kết nối chấp nhận kết nối
        // và trả về một socket mới mang tên socketForClient,
        // trong khi tcpListener tiếp tục nghe ngóng
        Socket socketForClient = tcpListener.AcceptSocket();
        if(socketForClient.Connected)
        {
            Console.WriteLine("Client connected");
            ClientHandler handler = new ClientHandler(socketForClient);
            handler.StartRead();
        }
    } // end for
} // end Run
}

*****

```

**Thí dụ 1-22: *Thiết đặt một client đối với asynchronous streaming network file server***

\*\*\*\*\*

```
using System;
using System.Net.Sockets;
using System.Threading;
using System.Text;
using System.Runtime.Serialization.Formatters.Binary;

public class AsynchNetWorkClient
{
    static public int Main()
    {
        AsynchNetWorkClient client = new AsynchNetWorkClient();
        return client.Run();
    }

    AsynchNetWorkClient() // hàm constructor
    {
        string serverName = "localhost";
        Console.WriteLine("Connecting to {0}", serverName);
        TcpClient tcpSocket = new TcpClient(serverName, 65000);
        streamToServer = tcpSocket.GetStream();
    }

    private int Run();
    {
        string message = @"D:\Test\Testtxt.txt";
        Console.WriteLine("Sending {0} to server", message);

        // tạo một streamWriter và dùng nó để viết message lên server
        System.IO.StreamWriter writer = new
            System.IO.StreamWriter(streamToServer);
        writer.WriteLine(message);
        writer.Flush();
        bool fQuit = false;

        // khi còn dữ liệu từ server, tiếp tục đọc
        while (!fQuit)
        {
            // buffer cầm giữ trả lời
            char[] buffer = new char[BufferSize];
            // Đọc câu trả lời
            System.IO.StreamReader reader = new
                System.IO.StreamReader(streamToServer);
            // xem bao nhiêu byte được tìm thấy ở buffer
            int bytesRead = reader.Read(buffer, 0, BufferSize);
            if (bytesRead == 0) // none? quit
                fQuit = true;
            else
            {
                // cho hiển thị một chuỗi
                string theString = new String(buffer);
                Console.WriteLine(theString);
            }
        }
        streamToServer.Close(); // dọn dẹp
        return 0;
    }
}
```

```

    } // end Run
    private const int BufferSize = 256;
    private NetworkStream streamToServer;
} // end AsynchNetworkClient
*****

```

## 1.5. Sản sinh hàng loạt (Serialization)

Như bạn đã thấy, namespace **System.IO** định nghĩa một số lớp cho phép bạn gửi dữ liệu nhị phân hoặc kiểu ký tự lên các thiết bị trữ tin (đĩa hoặc ký ức). Điều hiển bạn chưa biết là làm thế nào cất trữ những thể hiện của những lớp tự tạo (custom class) lên stream để đưa cất trữ lên đĩa chẳng hạn, rồi lôi ra đọc xuống lại từ chỗ trữ.

Theo .NET Framework, khi một đối tượng được chuyển thành stream ghi lên đĩa, thì những dữ liệu thành viên khác nhau phải được *serialized*. *Serialization*<sup>2</sup> là từ dùng mô tả tiến trình chuyển đổi trạng thái của một đối tượng thành một loạt bytes tuyến tính nối đuôi nhau ghi ra thành một byte stream. Đối tượng cũng sẽ được serialized khi được trữ lên căn cứ dữ liệu hoặc khi được “kèm cặp vào hàng ngũ” (marshaled) xuyên qua một phạm trù (context), miền ứng dụng (app domain), tiến trình (process) hoặc biên giới máy tính (machine boundary). Byte stream này chứa tất cả thông tin cần thiết để có thể tái tạo nguyên trạng (gọi là *deserialization*) của đối tượng dùng về sau. Dịch vụ serialization của .NET khá tinh vi: khi một đối tượng được serialized chuyển về stream, bất cứ những qui chiếu đối tượng được bổ sung về sau đòi hỏi đối tượng gốc phải được serialized lại. Thí dụ, khi một lớp được dẫn xuất, thì mỗi đối tượng thuộc một chuỗi kế thừa sẽ có khả năng viết ra trạng thái dữ liệu custom riêng của mình lên byte stream.

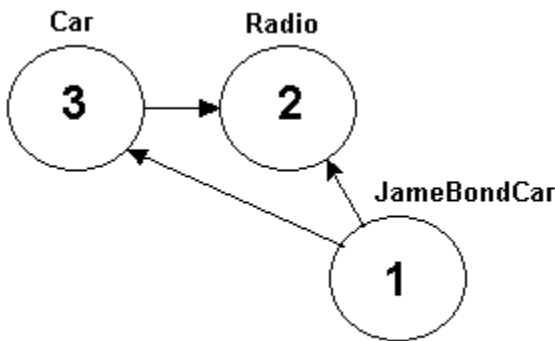
Một khi một lô đối tượng được trữ trên một stream, “mẫu dáng byte” (byte pattern) có thể được “tái định cư” (relocated) nếu thấy cần thiết. Thí dụ, bạn hình dung bạn đã serialized một stream những đối tượng lên một MemoryStream. Stream này có thể chuyển tới một máy tính nằm xa trên Windows clipboard, hoặc bị đốt cháy trên một CD, hoặc đơn giản ghi trữ lên một tập tin đĩa. Byte stream bất cần biết dữ liệu được trữ ở đâu. Điều nó quan tâm là dòng bit 0 và 1 tượng trưng một cách trung thực trạng thái của các đối tượng được serialized.

---

<sup>2</sup> Chúng tôi tạm dịch từ này là “sản sinh hàng loạt”.

## 1.5.1 Vai trò của Object Graph

CLR cung cấp việc hỗ trợ serialization thông qua một cái gọi là *Object Graph*, nghĩa là chuỗi những đối tượng liên hệ với nhau được serialized thành một stream có thể được qui chiếu một cách tập thể như là một object graph. *Object Graph* là một thể thức đơn giản cho phép lưu trữ cách một lô đối tượng được liên đới với nhau như thế nào. Muốn thiết lập những mối liên hệ giữa các đối tượng, một đối tượng được gán cho một con số duy nhất (số này rất hồ đồ không mang ý nghĩa gì với thế giới bên ngoài), theo sau là một graph của tất cả các item có liên hệ. Ta thử lấy một thí dụ đơn giản: giả sử bạn tạo một số lớp mô hình hóa các xe hơi, như theo hình 1-17. Bạn có lớp cao nhất Car có một liên hệ “has-a” (có một) với lớp Radio. Một lớp khác JamesBondCar, nói rộng lớp cơ bản Car. Hình 1-17 được gọi là object graph.



Hình 01-17: Object graph đơn giản

Trên hình 1-17 bạn có thể thấy lớp Car qui chiếu về lớp Radio (theo một liên hệ “has-a”) JamesBondCar qui chiếu về lớp Car (như là một subclass của nó) cũng như lớp Radio (như là kế thừa thành viên protected). Dựa trên mỗi qui chiếu đối tượng được gán một số hồ đồ, bạn có thể xây dựng một công thức sau đây:

```
[Car 3, ref 2], [Radio 2],  
[JamesBondCar 1, ref 3, ref 2]
```

Công thức trên được gọi là “mẫu dáng” (pattern) dùng serialized các đối tượng thành stream kèm theo trị của mỗi biến thành viên trên các lớp Car, Radio và JamesBondCar. Bạn có thể thấy là lớp Car có một lệ thuộc trên item 2 (đối tượng Radio). Ngoài ra, JamesBondCar có một lệ thuộc trên item 3 (đối tượng Car) cũng như trên item 2 (đối tượng Radio). Nếu bạn cho serialize một thể hiện của JamesBondCar thành một stream, object graph bảo đảm là các lớp Radio và Car cũng tham gia vào tiến trình serialization. Điều tốt lành là object graph được tự động thiết lập ở hậu trường bởi CLR.

## 1.5.2 Cấu hình các đối tượng cho việc sản sinh hàng loạt

Muốn một đối tượng sẵn sàng cho sản sinh hàng loạt, bạn cho đánh dấu mỗi lớp thông qua attribute **[Serializable]**. Chỉ có thể thôi. Nếu bạn xác định là trong một lớp nào đó có vài thành viên dữ liệu không tham gia vào trò chơi này, thì bạn cho đánh dấu các

vùng mục tin này với attribute **[NonSerialized]**. Điều này có thể hữu ích nếu bạn có biến thành viên (hoặc thuộc tính) trong một lớp serializable không cần được nhớ (chẳng hạn hằng, dữ liệu trung gian v.v.). Thí dụ 1-23 sau đây là lớp Radio, được đánh dấu là serializable (ngoại trừ một biến thành viên duy nhất):

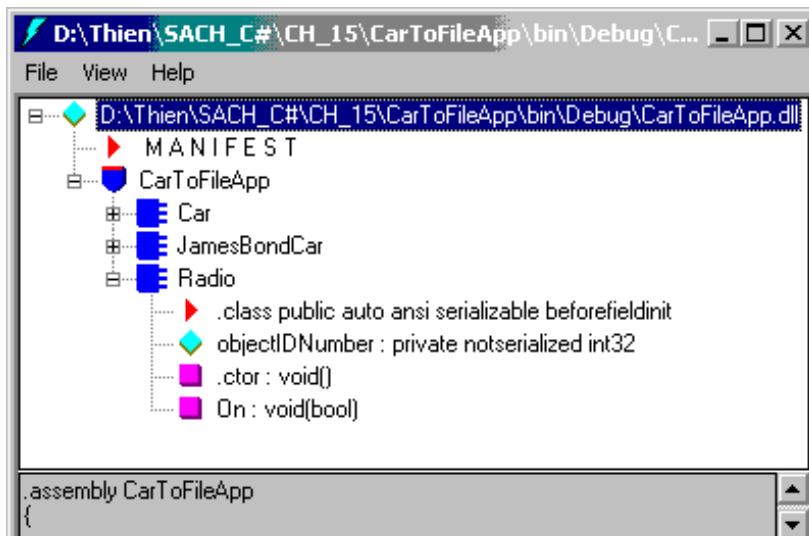
```
// Lớp Radio có thể tham gia vào .NET serialization scheme
[Serializable]
public class Radio
{
    // bạn không quan tâm đến việc cất trữ biến thành viên sau đây
    [NonSerialized]
    private int objectIDNumber = 9;

    public Radio() {}

    public void On(bool state)
    {
        if(state == true)
            MessageBox.Show("Music is on...");
        else
            MessageBox.Show("No tunes...");
    }
}
```

Những attributes được đánh dấu trên metadata của lớp; bạn có thể dùng trình tiện ích ILDasm.exe cho hiện lên để mà xem (hình 1-18).

Để kết thúc với việc lập trình cây đẳng cấp Car, sau đây là những định nghĩa đối với lớp cơ bản Car và lớp con JamesBond Car, mỗi lớp đều được đánh dấu bởi attribute [Serializable].



Hình 01-18: Các attribute Serializable và NonSerialized

**Thí dụ 1-23: Một lớp Car đơn giản serializable**

\*\*\*\*\*

```
// Lớp Car có thể serializable
[Serializable]
public class Car
{
    protected string petName; // tên cưng chiều
    protected int maxSpeed;    // tốc độ tối đa
    protected Radio theRadio = new Radio();

    public Car(string petName, int maxSpeed) // hàm constructor
    {
        this.petName = petName;
        this.maxSpeed = maxSpeed;
    }

    public Car() {} // hàm constructor không thông số

    public String PetName // hàm property
    {
        get { return petName; }
        set { petName = value; }
    }

    public int MaxSpeed
    {
        get { return maxSpeed; }
        set { maxSpeed = value; }
    }

    public void TurnOnRadio(bool state)
    {
        theRadio.On(state);
    }
}

// Lớp JamesBondCar cũng có thể serializable
[Serializable]
public class JamesBondCar: Car
{
    protected bool isFlightWorthy; // đáng bay không
    protected bool isSeaWorthy;    // đang chạy trên biển không

    public JamesBondCar() {} // hàm constructor

    public JamesBondCar(string petName, int maxSpeed, bool canFly,
        bool canSubmerge): base(petName, maxSpeed)
    {
        this.isFlightWorthy = canFly;
        this.isSeaWorthy = canSubmerge;
    }

    public void Fly()
    {
        if(isFlightWorthy)
            MessageBox.Show("Cất cánh!");
        else
            MessageBox.Show("Roi vào vách núi!");
    }

    public void GoUnderWater()
```

```

{   if (isSeaWorthy)
        MessageBox.Show("Lặn dưới biển...");
    else
        MessageBox.Show("Chìm xuống đáy biển");
}
*****

```

### 1.5.2.1 Sử dụng một Formatter

Một khi bạn đã cấu hình các lớp của mình tham gia vào việc sản sinh hàng loạt của .NET, bước kế tiếp là chọn một dạng thức (format) nào dùng để tồn lưu lâu dài (persistent) object graph của bạn. Khi dữ liệu được serialized, nó sẽ được đọc hoặc bởi cùng chương trình hoặc bởi một chương trình khác trên một máy khác. Trong bất cứ trường hợp nào, đoạn mã đọc dữ liệu sẽ chờ đợi là dữ liệu ở một dạng thức nào đó. Trong đa số trường hợp trên một ứng dụng .NET, dạng thức được chờ đợi là hoặc dạng nhị phân nguyên sinh hoặc SOAP (Simple Object Access Protocol). Do đó, namespace **System.Runtime.Serialization.Formatters** chứa hai formatter mặc nhiên (\*.Binary và \*.Soap).

**Bạn để ý** SOAP là một nghi thức đơn giản “nhẹ cân” dựa trên XML để trao đổi thông tin xuyên Web. SOAP mang tính đơn thể (modular) khá cao và có khả năng mở rộng. Nó nâng cao khả năng các công nghệ Internet hiện hữu như HTTP và SMTP.

Như bạn có thể thấy lớp **BinaryFormatter** sẽ serialize object graph của bạn thành một stream sử dụng đến dạng thức nhị phân cô đặc (compact binary format); lớp này rất hữu ích đối với chỗ trữ nhanh tại chỗ. Còn lớp **SoapFormatter** sẽ tượng trưng object graph của bạn như là một thông điệp SOAP (biểu diễn theo dạng thức XML), và thường được dùng trên Internet. Các lớp formatter sẽ thiết đặt giao diện **IFormatter** và **ISerializable**.

Muốn serialize các đối tượng theo dạng thức nhị phân, bạn chỉ cần khai báo chỉ thị using như sau:

```

// Cần gọi đối tượng theo dạng thức nhị phân
using System.Runtime.Serialization.Formatter.Binary;

```

Tuy nhiên, lớp **SoapFormatter** được định nghĩa trong một assembly riêng biệt. Muốn dùng SOAP Formatter bạn phải bắt đầu đặt để qui chiếu về **System.Runtime.Serialization.Formatter.Soap.dll**, rồi dùng chỉ thị using sau đây:

```

// Cần gọi đối tượng theo dạng thức SOAP
using System.Runtime.Serialization.Formatter.Soap;

```



### 1.5.2.2 *Vai trò của namespace* *System.Runtime.Serialization*

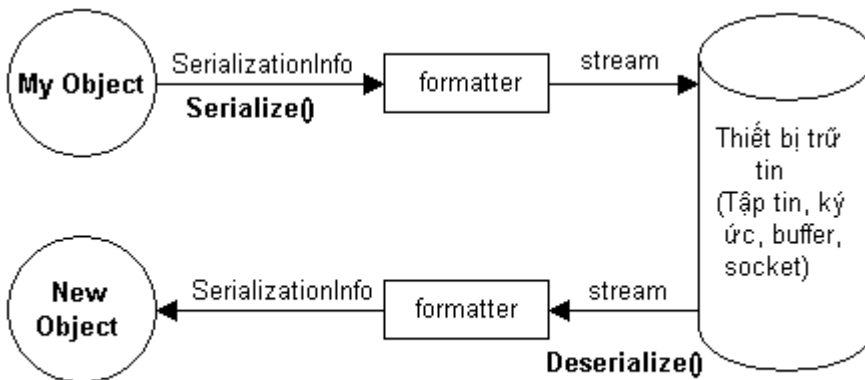
Nếu một ngày nào đó, bạn nổi hứng muốn tạo một custom formatter, thì bạn sẽ cần đến một số lớp được định nghĩa trong namespace **System.Runtime.Serialization**. Ngoài ra, nếu bạn muốn cấu hình các đối tượng của bạn sử dụng đến custom serialization, thì những lớp này cũng đáng quan tâm. Mặc dù, việc tạo custom formatter là ngoài phạm vi tập sách này, bảng 1-15 liệt kê một số lớp (không phải toàn bộ) cốt lõi mà bạn có thể quan tâm đến:

**Bảng 1-15: Các lớp cốt lõi của System.Runtime.Serialization.**

Tên các lớp	Ý nghĩa
<b>Formatter</b>	Một lớp abstract cung cấp chức năng cơ bản đối với những runtime serialization formatter.
<b>ObjectIDGenerator</b>	Lớp này cho kết sinh mã nhận diện ID đối với những đối tượng trong một object graph.
<b>ObjectManager</b>	Lớp này theo dõi các đối tượng khi chúng đang bị deserialized.
<b>SerializationBinder</b>	Một lớp cơ bản abstract cung cấp chức năng để serialize một lớp biến thành stream.
<b>SerializationInfo</b>	Lớp này được dùng bởi những đối tượng có hành vi custom serialization. SerializationInfo giữ lại với nhau tất cả các dữ liệu cần thiết cho serialize và deserialize một đối tượng. Thực chất, lớp này là một “túi thuộc tính” (property bag) cho phép thiết lập những cặp name/value tượng trưng cho một trạng thái của một đối tượng.

Cho dù bạn chọn loại formatter gì đi nữa, thì formatter vẫn phải lo việc chuyển tất cả thông tin cần thiết làm cho đối tượng thành bền vững trong tiến trình sản sinh hàng loạt. Các thông tin cần thiết bao gồm tên trọn vẹn đối tượng (chẳng hạn MyProject.MyClass.Foo), tên assembly chứa đối tượng (chẳng hạn tên thân hữu, phiên bản, và tùy chọn một tên mạnh) và kể cả bất cứ thông tin tình trạng tượng trưng bởi lớp **SerializationInfo**.

Trong tiến trình deserialization, formatter sử dụng đến thông tin này để tạo một bản sao y chang của đối tượng, sử dụng đến thông tin trích từ stream nằm ẩn sau. Bạn có thể hình dung tiến trình serialization và deserialization như theo hình 1-19:



Hình 01-19: Tiến trình serialization

### 1.5.3 Sản sinh hàng loạt sử dụng đến Binary Formatter

Bạn nhớ lại lớp **BinaryFormatter** thuộc namespace **System.Runtime.Serialization.Formatter.Binary**, và định nghĩa hai hàm hành sự cốt lõi: **Serialize()** và **Deserialize()**.

Hàm **Serialize()** lo serialize một đối tượng hoặc một object graph thành một stream. Ngược lại, hàm **Deserialize()** lo giải serialization một byte stream thành một object graph.

Muốn biết sản sinh hàng loạt làm việc thế nào, bạn cần một lớp đơn giản mà bạn có thể serialize và deserialize. Bạn có thể bắt đầu tạo một lớp cho mang tên **SumOf** (tổng cộng của). **SumOf** có 3 biến thành viên private:

```
private int startNumber = 1;
private int endNumber;
private int[] theSums;
```

Biến bản dãy **theSums** tượng trưng trị tổng của tất cả các số từ **startNumber** đến **endNumber**. Do đó, nếu **startNumber** là 1 và **endNumber** là 10, thì bản dãy sẽ có những trị: 1, 3, 6, 10, 15, 21, 28, 36, 45, 55. Mỗi trị là tổng của trị trước cộng với trị kế tiếp trên loạt số.

Hàm constructor nhận hai số nguyên: số khởi đi, **startNumber**, và số kết thúc, **endNumber**. Hàm gán các số này cho biến cục bộ và gọi một hàm hỗ trợ để tính nội dung bản dãy:

```
public SumOf(int start, int end)
{ startNumber = start;
```

```
        endNumber = end;  
        ComputeSums();  
    }
```

Hàm hỗ trợ **ComputeSums()** sẽ điền vào nội dung bản dãy bằng cách tính những tổng trong loạt số từ **startNumber** đến **endNumber**:

```
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];  
    }  
}
```

Vào bất cứ lúc nào, bạn có thể hiển thị nội dung bản dãy sử dụng đến vòng lặp **foreach**:

```
private void DisplaySums()  
{  
    foreach (int i in theSums)  
    {  
        Console.WriteLine("{0}, ", i);  
    }  
}
```

### 1.5.3.1 Cho serialize đối tượng

Bây giờ cho đánh dấu lớp là có thể làm sản sinh hàng loạt thông qua attribute **[Serializable]**:

```
[Serializable]  
class SumOf
```

Muốn triệu gọi sản sinh hàng loạt, trước tiên, bạn cần một đối tượng file stream mà bạn sẽ thực hiện sản sinh hàng loạt trên ấy đối với đối tượng **SumOf**.

```
FileStream fileStream = new  
    FileStream("DoSum.out", FileMode.Create);
```

Bây giờ bạn sẵn sàng triệu gọi hàm **Serialize()** của formatter, trao qua hàm đối tượng stream và đối tượng **SumOf** cần serialize. Vì việc này xảy ra trong hàm hành sự của **SumOf**, bạn có thể trao đối tượng **this**, chỉ về đối tượng hiện hành:

```
BinaryFormatter.Serialize(fileStream, this);
```

Như vậy đối tượng SumOf sẽ được serialized lên đĩa.

### 1.5.3.2 Cho deserialize đối tượng

Muốn tái tạo lại đối tượng, bạn cho mở tập tin và yêu cầu một binary formatter Deserialize tập tin:

```
private static SumOf DeSerialize()
{
    FileStream fileStream = new FileStream(
        "DoSum.out", FileMode.Open);
    BinaryFormatter binaryFormatter = new BinaryFormatter();
    return (SumOf) binaryFormatter.Deserialize(fileStream);
    fileStream.Close();
} // end DeSerialize
```

Để bảo đảm là mọi việc sẽ chạy tốt, trước tiên, bạn cho hiển lộ một đối tượng mới kiểu SumOf và yêu cầu nó tự serialize nó và bảo nó hiển thị trị:

```
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();
} // end Main
```

Thí dụ 1-24 liệt kê toàn bộ đoạn mã minh họa việc serialization và deserialization

#### **Thí dụ 1-24: Cho Serialization và Deserialization một đối tượng**

\*\*\*\*\*

```
namespace Prog_CSharp
{
    using System;
    using System.IO;
    using System.Runtime.Serialization;
    using System.Runtime.Serialization.Formatters.Binary;

    [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();
        } // end Main

        public SumOf(int start, int end)
```

```

    {   startNumber = start;
        endNumber = end;
        ComputeSums();
        DisplaySums();
        Serialize()
    }   // end SumOf

    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];
        }
    }   // end ComputeSums

    private void DisplaySums()
    {   foreach (int i in theSums)
        {   Console.WriteLine("{0}, ", i);
        }
    }   // end DisplaySums

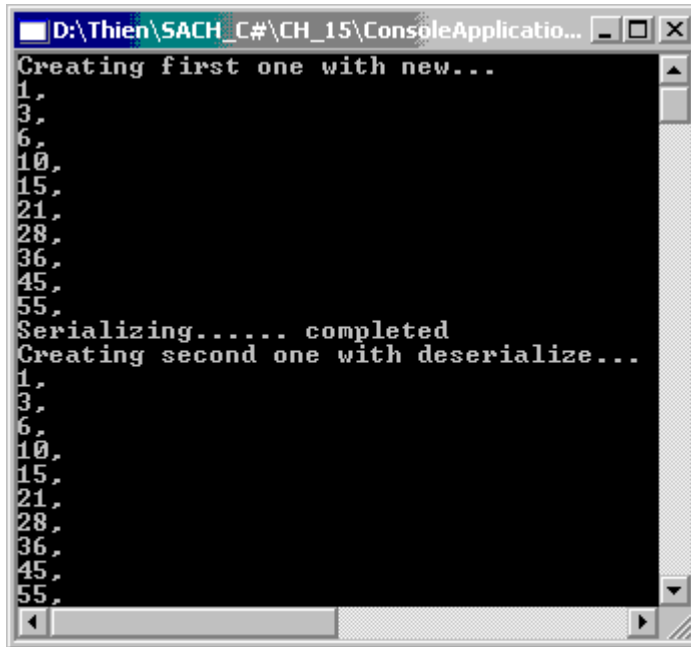
    private void Serialize()
    {   Console.Write("Serializing...");
        FileStream fileStream = new FileStream(
            @"D:\Test\DoSum.out", FileMode.Create);
        BinaryFormatter binaryFormatter = new BinaryFormatter();
        binaryFormatter.Serialize(fileStream, this);
        Console.WriteLine("... completed");
        fileStream.Close();
    }   // end Serialize

    private static SumOf DeSerialize()
    {   FileStream fileStream = new FileStream(
        @"D:\Test\DoSum.out", FileMode.Open);
        BinaryFormatter binaryFormatter = new BinaryFormatter();
        return (SumOf) binaryFormatter.Deserialize(fileStream);
        fileStream.Close();
    }   // end DeSerialize

    private int startNumber = 1;
    private int endNumber;
    private int[] theSums;
}   // end class SumOf
}   // end Prog_CSharp

```

Hình 1-20 là kết xuất của chương trình trên cho thấy đối tượng được tạo ra, hiển thị rồi serialized. Sau đó, đối tượng lại được deserialized và cho kết xuất lại lên màn hình, không có mất mát.



Hình 01-20: Serialization và Deserialization  
một đối tượng

## 1.5.4 Sản sinh hàng loạt sử dụng đến SOAP Formatter

Muốn sử dụng đến SOAP Formatter, bạn cần đặt để một qui chiếu về assembly chứa đựng **System.Runtime.Serialization.Formatter.Soap.dll**. Đoạn mã sau đây nói rộng thí dụ sản sinh hàng loạt trước để ghi bèn vững JamesBondCar sử dụng đến lớp **SoapFormatter**.

```
using System.Runtime.Serialization.Formatters.Soap;

public static void Main()
{
    // tạo đối tượng Car và hoạt động với đối tượng này
    JamesBondCar myAuto = new JamesBondCar("Fred", 50, false, true);
    myAuto.TurnOnRadio(true);
    myAuto.GoUnderWater();
    // tạo một file stream
    FileStream myStream = File.Create("CarData.xml");
    // serialization
    SoapFormatter myXMLFormat = new SoapFormatter();
    myXMLFormat.Serialize(myStream, myAuto);
    myStream.Close();
}
```

```

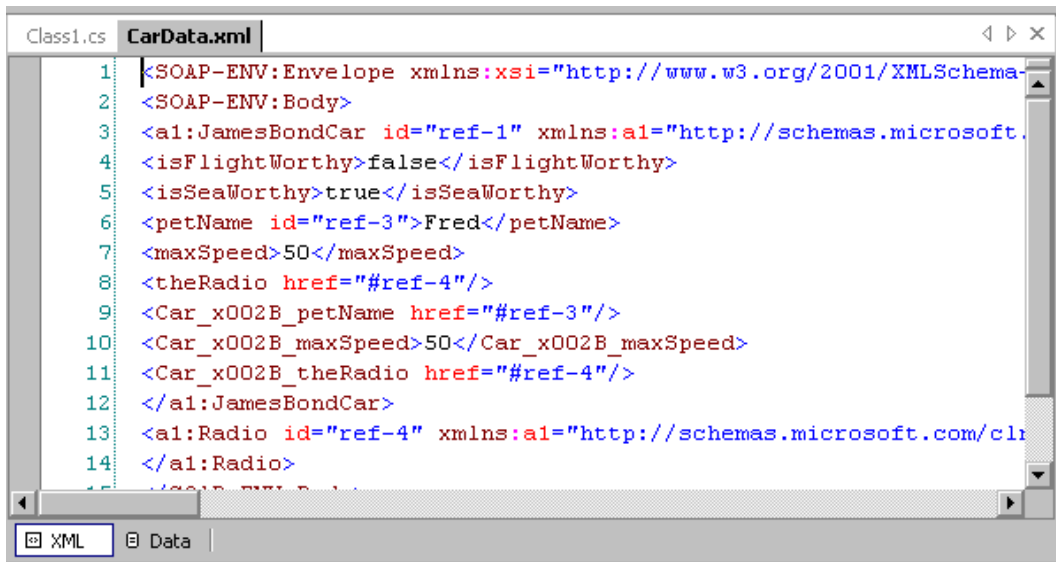
// đọc vào CarData.dat
myStream = File.OpenRead("CarData.xml");
JamesBondCar carFromXML =
    (JamesBondCar)myXMLFormat.Deserialize(myStream);
Console.WriteLine(carFromXML.PetName + " is alive!");
carFromXML.TurnOnRadio(true);
myStream.Close();
}

```

Như bạn có thể thấy lớp SoapFormatter có cùng giao diện như với **BinaryFormatter**. Bạn sử dụng đến **Serialize()** và **Deserialize()** để chuyển qua lại object graph với stream. Bạn thử dump tập tin \*.xml ra thì sẽ thấy hình thù ra sao. Hình 1-21:

## 1.6 Thụ lý dữ liệu vãng lai

Trong chừng mực nào đó, cách tiếp cận để thực hiện sản sinh hàng loạt như theo thí dụ 1-24 xem ra “vô duyên”. Vì bạn có thể tính nội dung của bản dây khi có số đầu và số cuối, thì không có lý do gì phải chờ đợi các phần tử lên đĩa. Mặc dù việc tính toán trên không tốn kém chi về thời gian vì đây là một bản dây nhỏ, nhưng có thể trở thành tốn hao thời gian khi gặp phải một bản dây lớn.



Hình 01-21: JamesBondCar được serialized sử dụng một SoapFormatter XML

Như đã nói, nếu bạn xác định là trong một lớp nào đó có vài thành viên dữ liệu không tham gia vào trò chơi sản sinh hàng loạt này, thì bạn cho đánh dấu các vùng mục tin này với attribute **[NonSerialized]**.

```
[NonSerialized].
private int[] theSums;
```

Tuy nhiên, nếu bạn không serialize bản dãy, thì đối tượng mà bạn tạo ra sẽ không chính xác khi bạn deserialize bản dãy. Bản dãy sẽ trống rỗng. Bạn nhớ lại, khi bạn deserialize đối tượng đơn giản là bạn đọc từ dạng thức được serialize; không hàm hành sự nào được chạy.

Để sửa chữa đối tượng trước khi bạn trả nó về cho hàm triệu gọi, bạn cho thiết đặt giao diện **IDeserializationCallback**:

```
[Serialization]
class SumOf: IDeserializationCallback
```

Ngoài ra, cũng cho thiết đặt luôn một hàm duy nhất của giao diện này: **OnDeserialization()**. CLR hứa hẹn là nếu bạn thiết đặt giao diện này, hàm **OnDeserialization()** trên lớp của bạn sẽ được triệu gọi khi trọn object graph được deserialize. Chính đây là điều bạn mong muốn. CLR sẽ tái tạo những gì bạn serialize, lúc này bạn có cơ may sửa chữa phần nào không được serialize.

Việc thiết đặt khá đơn giản; chỉ cần yêu cầu đối tượng tính toán lại loạt số:

```
public virtual void OnDeserialization(Object sender)
{   ComputeSums();
}
```

Bằng cách không cho serialize bản dãy, bạn làm cho việc deserialization chậm hơn vì phải lấy thời gian tính lại bản dãy, nhưng được cái là tập tin nhỏ hơn. Bạn thử chạy một chương trình với số từ 1 đến 5000. Trước khi cho đặt **[NonSerialized]** trên bản dãy, tập tin được serialize chiếm khoảng 20K. Nhưng sau khi cho đặt **[NonSerialized]** thì tập tin còn lại khoảng 1K. Cũng khá đấy.

Thí dụ 1-25 sau đây liệt kê toàn bộ mã nguồn sử dụng số từ 1 đến 5 để cho “nhẹ cân” khi kết xuất:

### ***Thí dụ 1-25: Làm việc với một đối tượng nonserialized***

```
*****
namespace Prog_CSharp
{
    using System;
    using System.IO;
```



```

using System.Runtime.Serialization;
using System.Runtime.Serialization.Formatters.Binary;

[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();
    } // end Main

    public SumOf(int start, int end)
    {
        startNumber = start;
        endNumber = end;
        ComputeSums();
        DisplaySums();
        Serialize();
    } // end SumOf

    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];
        }
    } // end ComputeSums

    private void DisplaySums()
    {
        foreach (int i in theSums)
        {
            Console.WriteLine("{0}, ", i);
        }
    } // end DisplaySums

    private void Serialize()
    {
        Console.WriteLine("Serializing...");
        FileStream fileStream = new FileStream(
            @"D:\Test\DoSum.out", FileMode.Create);
        BinaryFormatter binaryFormatter = new BinaryFormatter();
        binaryFormatter.Serialize(fileStream, this);
        Console.WriteLine("... completed");
        fileStream.Close();
    } // end Serialize

    private static SumOf DeSerialize()
    {
        FileStream fileStream = new FileStream(
            @"D:\Test\DoSum.out", FileMode.Open);
        BinaryFormatter binaryFormatter = new BinaryFormatter();
        return (SumOf) binaryFormatter.Deserialize(fileStream);
        fileStream.Close();
    } // end DeSerialize

```

```
// sửa chữa dữ liệu nondeserialized
public virtual void OnDeserialization (Object sender)
{   ComputeSums();

}
private int startNumber = 1;
private int endNumber;
[NonSerialized] private int[] theSums;
} // end class SumOf
} // end Prog_CSharp
*****
```

Nếu bạn cho chạy lại chương trình này thì kết quả giống như trên hình 1-19. Bạn đã thành công trong việc sản sinh hàng loạt lên đĩa và tái tạo lại bằng cách deserialization.

## 1.7 Isolated Storage

.NET CLR cung cấp *isolated storage* (ký ức trữ cách ly) cho phép lập trình viên trữ dữ liệu trên căn bản người sử dụng (per-user basis). Isolated storage cung cấp chức năng tương tự như các tập tin .ini trên Windows, hoặc như key HKEY\_CURRENT\_USER trên Windows Registry.

Các ứng dụng sẽ cất trữ dữ liệu trên một *data compartment* được gắn liền với ứng dụng. Data compartment là một khái niệm trừu tượng, không ám chỉ một vùng ký ức nào cụ thể; nó bao gồm một hoặc nhiều tập tin isolated storage, được gọi là *store*, chứa vị trí hiện thời của thư mục nơi chứa thực thụ dữ liệu. Do đó, CLR sẽ thiết đặt data compartment với một data store điển hình là một thư mục trên file system.

Viên quản lý được tự do ấn định giới hạn mỗi ứng dụng có thể dùng riêng cho mình bao nhiêu isolated storage, cũng như áp dụng chế độ an ninh làm thế nào đoạn mã không mấy tin tưởng không được triệu gọi đoạn mã nhiều tin tưởng hơn để viết isolated storage.

Điểm quan trọng đối với isolated storage là CLR cung cấp một chỗ chuẩn để trữ dữ liệu của ứng dụng, và không bắt buộc (hoặc hỗ trợ) một sơ đồ bố trí các vùng mục tin (data layout) hoặc cú pháp đối với dữ liệu này. Nói tóm lại, bạn có thể trữ bất cứ cái gì bạn thích trên isolated storage.

Điển hình là bạn sẽ trữ dữ liệu kiểu văn bản, hoặc thường xuyên bạn trữ những cặp name/value. Isolated storage là một cơ chế rất thuận tiện để trữ những thông tin cấu hình của người sử dụng, chẳng hạn tên login, vị trí của những cửa sổ hoặc ô control khác nhau, cũng như thông tin đặc biệt liên quan đến các ứng dụng hoặc người sử dụng. Dữ liệu sẽ được trữ trên một tập tin riêng biệt cho mỗi người sử dụng, nhưng các tập tin có thể được

cách ly sâu hơn để phân biệt những khía cạnh khác nhau dựa trên “lý lịch” của đoạn mã (theo assembly hoặc theo domain ứng dụng).

Sử dụng isolated storage cũng khá đơn giản. Muốn viết một isolated storage, bạn bắt đầu tạo một thể hiện của lớp **IsolatedStorageFileStream**, mà bạn khởi gán với tên tập tin và một thể thức tạo tập tin (file mode) như create, append, v.v...:

```
IsolatedStorageFileStream configFile = new
    IsolatedStorageFileStream("Tester.cfg", FileMode.Create);
```

Sau đó, bạn tạo một đối tượng **StreamWriter** trên tập tin này:

```
StreamWriter writer = new StreamWriter(configFile);
```

Tiếp theo, bạn viết lên stream này như bạn đã làm với bất cứ stream nào. Thí dụ 1-26 sau đây:

### ***Thí dụ 1-26: Làm việc với một isolated storage***

```
*****
namespace Prog_CSharp
{
    using System;
    using System.IO;
    using System.IO.IsolatedStorage;

    public class Tester
    {
        public static void Main()
        {
            Tester app = new Tester();
            app.Run();
        } // end Main

        private void Run()
        {
            // tạo cấu hình file stream
            IsolatedStorageFileStream configFile = new
                IsolatedStorageFileStream("Tester.cfg", FileMode.Create);
            // tạo một writer để viết lên stream
            StreamWriter writer = new StreamWriter(configFile);
            // viết vài dữ liệu lên 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);
            // flush buffer và dọn dẹp
            writer.Flush();
            writer.Close();
            configFile.Close();
        } // end Run
    } // end Tester
}
```

```
} // Prog_CSharp
*****
```

Sau khi cho chạy đoạn mã kể trên, bạn cho truy tìm tập tin Tester.cfg trên ổ đĩa chứa hệ điều hành WINNT, bạn sẽ thấy tập tin này nằm ở thư mục như sau (trên máy của chúng tôi):

```
C:\Documents and Settings\Administrator\Local Settings\Application
Data\IsolatedStorage\fm53b2aporaplusrfxmaxsq\
Url.dzgo2yfvysrkewwdamkpg2d12y534ri\
Url.dzgo2yfvysrkewwdamkpg2d12y534ri\Files
Size: 58 bytes
Type: CFG File
Modified: 8/17/2003 9:04 AM
```

Và nếu bạn dùng NotePad đọc tập tin này bạn sẽ thấy nội dung như sau:

```
Last access: 8/17/2003 9:04:19 AM
Last position = 27,35
```

Hoặc bạn có thể truy xuất dữ liệu này thông qua chương trình. Muốn thế, bạn cho mở lại tập tin này:

```
IsolatedStorageFileStream configFile = new
    IsolatedStorageFileStream("Tester.cfg", FileMode.Open);
```

Rồi tạo một đối tượng StreamReader:

```
StreamReader reader = new StreamReader(configFile);
```

rồi dùng vòng lặp do...while để đọc tập tin

```
string theEntry;
do
{
    theEntry = reader.ReadLine();
    Console.WriteLine(theEntry);
} while (theEntry != null)
```

Thí dụ 1-27 là toàn bộ chương trình đọc tập tin Tester.cfg.

### ***Thí dụ 1-27: Đọc từ isolated storage***

```
*****
namespace Prog_CSharp
{
    using System;
    using System.IO;
    using System.IO.IsolatedStorage;
```

```

public class Tester
{
    public static void Main()
    {
        Tester app = new Tester();
        app.Run();
    } // end Main

    private void Run()
    {
        // mở tập tin Tester.cfg
        IsolatedStorageFileStream configFile = new
            IsolatedStorageFileStream("Tester.cfg", FileMode.Open);
        // tạo một reader để đọc lên stream
        StreamReader reader = new StreamReader(configFile);
        // đọc dữ liệu của tập tin Tester.cfg.
        string theEntry;
        do
        {
            theEntry = reader.ReadLine();
            Console.WriteLine(theEntry);
        } while (theEntry != null);

        // dọn dẹp
        reader.Close();
        configFile.Close();
    } // end Run
} // end Tester
} // Prog_CSharp
*****

```

### Kết xuất

Last access: 8/17/2003 9:04:19 AM

Last position = 27,35

## 1.8 Custom Serialization và giao diện ISerializable

Tiếp cận mặc nhiên để cho trử bền vững một kiểu dữ liệu “cây nhà lá vườn” (custom) rất đơn giản: cho đánh dấu một lớp với attribute [Serializable]. Khi một formatter được trao object graph, thì tất cả các đối tượng được qui chiếu sẽ được chuyển cho stream. Nếu đây diễn hình đúng là cách ứng xử bạn mong muốn, thì namespace **System.Runtime.Serialization** sẽ cung cấp những thể thức cho phép bạn customize tiến trình sản sinh hàng loạt.

Khi bạn muốn tham gia vào trò chơi sản sinh hàng loạt, bước đầu tiên phải làm là cho thi công giao diện chuẩn **ISerializable** lên lớp mà ta sẽ dùng custom serialization, như sau:

```
// khi bạn muốn tham gia tiến trình serialization,  
// thì cho thi công giao diện ISerializable  
public interface ISerializable  
{ public virtual void GetObjectData(SerializationInfo info,  
                                     StreamingContext context);  
}
```

Giao diện này định nghĩa một hàm hành sự duy nhất **GetObjectData()**, hàm này được triệu gọi bởi formatter trong tiến trình sản sinh hàng loạt. Việc thi công hàm này sẽ điền đầy thông số nhập **SerializationInfo** bởi một loạt cặp “tên-trị”. Kiểu dữ liệu **SerializationInfo** thực chất là một “property bag” (bao chứa các thuộc tính) khá quen thuộc đối với lập trình viên COM.

Ngoài việc thi công giao diện **ISerializable**, tất cả các đối tượng thi công custom serialization phải cung cấp một hàm constructor khá đặc biệt mang dấu ẩn như sau:

```
// bạn phải cung cấp một hàm custom constructor với dấu ẩn như sau  
// cho phép runtime engine đặt để trạng thái của đối tượng của bạn  
class SomeClass  
{  
    private SomeClass (SerializationInfo si, StreamingContext ctx) {...}  
}
```

Bạn để ý, tầm hoạt động của hàm này là private. Điều này bảo đảm là một người sử dụng thông thường không bao giờ tạo một đối tượng theo thể thức này.

Như bạn có thể thấy, thông số đầu tiên của hàm là một thể hiện của kiểu dữ liệu **SerializationInfo**, cho phép bạn cấu hình một loạt các cặp name/value tượng trưng cho trạng thái của đối tượng của bạn. **SerializationInfo** định nghĩa một hàm thành viên **AddValue()** được nạp chồng nhiều lần cho phép bạn khai báo bất cứ kiểu dữ liệu nào (string, integer, float, boolean, v.v...). Ngoài ra, một số hàm hành sự **GetXXXX()** cũng được cung cấp để trích thông tin từ kiểu dữ liệu **SerializationInfo** để điền vào các biến thành viên của đối tượng. Bạn sẽ thấy trong chốc lát các biến này hoạt động thế nào.

Thông số thứ hai, là kiểu dữ liệu **StreamingContext** chứa thông tin liên quan đến nguồn hoặc đích dưới dạng bit. Hàm thành viên của kiểu dữ liệu này thuộc tính **State**, tượng trưng cho một trị enumeration **StreamingContextStates**, như theo bảng 1-16.

**Bảng 1-16: Các thành viên của Enumeration *StreamingContextStates***

Các thành viên	Mô tả
<b>All</b>	Cho biết dữ liệu được serialized có thể được chuyển đi đi về về từ bất cứ phạm trù (context) khác.
<b>Clone</b>	Cho biết object clone được sao y chang.
<b>CrossAppDomain</b>	Cho biết source context hoặc destination context là một AppDomain mới.
<b>CrossMachine</b>	Cho biết source context hoặc destination context là một máy khác.
<b>CrossProess</b>	Cho biết source context hoặc destination context là một process khác.
<b>File</b>	Cho biết source context hoặc destination context là một tập tin.
<b>Other</b>	Cho biết phạm trù sản sinh hàng loạt là không biết được.
<b>Persistence</b>	Cho biết source context hoặc destination context là một kho trữ bền vững (nghĩa là được cất trữ lâu dài). Đây có thể là những căn cứ dữ liệu, các tập tin hoặc kho trữ phòng hồ (backing store). Người sử dụng phải giả định là dữ liệu bền vững “sống” lâu dài hơn tiến trình đã tạo ra dữ liệu này và không serialize các đối tượng theo cách thức nào đó mà deserialization đòi hỏi truy xuất bất cứ dữ liệu nào từ tiến trình hiện hành.
<b>Remoting</b>	Cho biết source context hoặc destination context ở một vị trí từ xa không biết được. Người sử dụng không thể giả định liệu xem trên cùng một máy hay không.

## 1.8.1 Một thí dụ đơn giản

Xin nhắc lại là bạn khỏi cần bỏ ngang cơ chế sản sinh hàng loạt mặc nhiên mà .NET Runtime cung cấp. Tuy nhiên, để minh họa, sau đây là một phiên bản kiểu dữ liệu Car được cấu hình tham gia vào custom serialization. Bạn sẽ không làm gì đặc biệt trong việc thi công **GetObjectState()** hoặc hàm custom constructor. Thay vì đó, mỗi hàm hành sự sẽ tuân xỏ (dump out) các thông tin liên quan đến phạm trù hiện hành và thao tác kiểu dữ liệu nhập **SerializationInfo**. Sau đây là bảng liệt in 1-28.

**Thí dụ 1-28: Lớp *CustomCarType***

```

*****
using System.Runtime.Serialization;
using System.Runtime.Serialization.Formatters.Binary;

...

[Serializable]
public class CustomCarType: ISerializable
{
    public string petName;

```

```

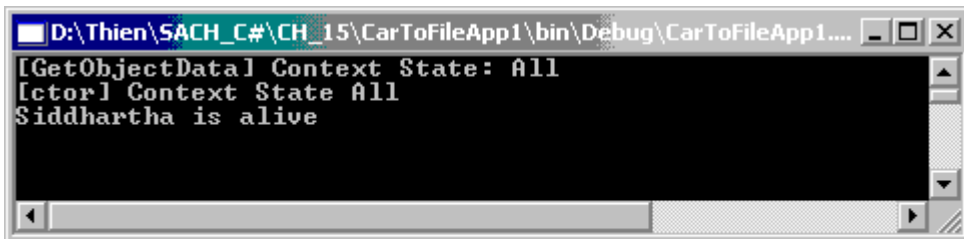
        public int maxSpeed;
        public CustomCarType(string s, int i)
        {petName = s; maxSpeed = i;}

        public void GetObjectData(SerializationInfo si,
                                   StreamingContext ctx)
        { Console.WriteLine("[GetObjectData] Context State: {0}",
                             ctx.State.ToString());
          si.AddValue("CapPetName", petName);
          si.AddValue("maxSpeed", maxSpeed);
        }

        private CustomCarType (SerializationInfo si,
                                StreamingContext ctx)
        { Console.WriteLine("[ctor] Context State {0}",
                             ctx.State.ToString());
          petName = si.GetString("CapPetName");
          maxSpeed = si.GetInt32("maxSpeed");
        }
    }
}
*****

```

Bây giờ kiểu dữ liệu CustomCarType đã được cấu hình xong với cấu trúc đúng đắn, bạn sẽ thấy là tiến trình serialization và deserialization không thay đổi, như theo liệt in 1-29 và hình 1-22:



**Hình 01-22: Kết quả của custom serialization**

### ***Thí dụ 1-29: Serialization & Deserialization một custom serialization***

```

*****
public static int Main(string[] args)
{
    CustomCarType myAuto = new CustomCarType("Siddhartha", 50);
    Stream myStream = File.Create(
        @"D:\Thien\SACH_C#\CH_15\CarData1.dat");
    BinaryFormatter myBinaryFormat = new BinaryFormatter();
    myBinaryFormat.Serialize(myStream, myAuto);
    myStream.Close();
    myStream = File.OpenRead(@"D:\Thien\SACH_C#\CH_15\CarData1.dat");
    CustomCarType carFromDisk =
        (CustomCarType)myBinaryFormat.Deserialize(myStream);
    Console.WriteLine(carFromDisk.petName + " is alive");
}

```



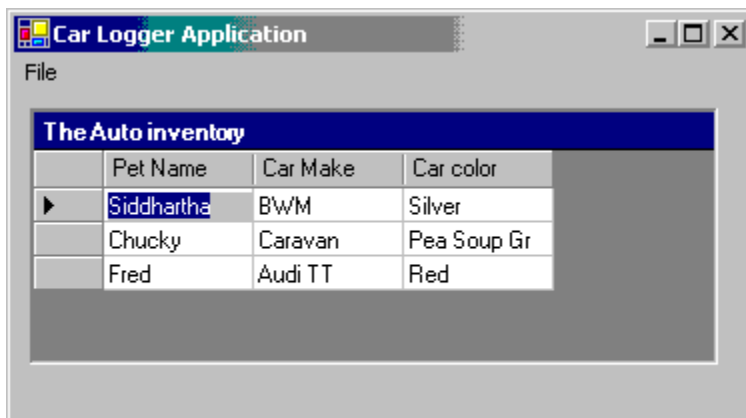
```

        Console.ReadLine();
        return 0;
    }
    *****

```

## 1.9 Một ứng dụng Windows Forms: Car Logger

Để kết thúc chương này, chúng tôi đưa ra một ứng dụng Windows Forms sử dụng đến những kỹ thuật mà chúng tôi đã đề cập qua. Ứng dụng Car Logger cho phép người sử dụng tạo một tập tin kiểm kê tồn kho các chiếc xe kiểu dữ liệu Car (được trữ trên một ArrayList) và được hiển thị trên một ô control Windows Forms, DataGrid, khác (hình 1-23). Vì chúng tôi chỉ tập trung đề cập đến logic của sản sinh hàng loạt, nên chỉ cho DataGrid này mang tính đọc mà thôi (read-only).



Hình 01-23: Ứng dụng Car Logger

Trình đơn **File** có 4 mục chọn, ngoài Exit, cho phép thao tác tên ArrayList nằm đằng sau. Bảng 1-17 mô tả những mục chọn này:

**Bảng 1-17: Trình đơn File với các mục chọn**

Mục chọn	Name	Mô tả
Clear All Cars	menuItemClear	Cho xóa trắng ArrayList và refresh DataGrid.
Exit	menuItemExit	Thoát khỏi ứng dụng
Make New Car	menuItemNewCar	Cho hiển thị một khung đối thoại custom cho phép người sử dụng khổ vào dữ liệu của một chiếc xe mới rồi refresh DataGrid.

Open Car File	nenuItemOpen	Cho phép người sử dụng mở một tập tin *.car rồi refresh DataGridView. Tập tin là kết quả của BinaryFormatter.
Save Car File	menuItemSave	Cho cất trữ tất cả các đối tượng Car trên DataGridView thành một tập tin *.car.

### ***Tạo khung sườn ứng dụng Windows Forms***

Trước tiên, bạn tạo một ứng dụng Windows Forms bằng cách ra lệnh **File | New | Project**, để cho hiện lên khung đối thoại **New Project**, rồi chọn **Visual C#** ở khung **Project Types** bên trái và **Windows Application** trên khung **Template** bên phải. Tiếp theo bạn khở vào tên ứng dụng ở ô Name, và lối tìm về ô Location, rồi ấn nút <OK>. Chúng tôi cho đặt tên ứng dụng là **CarLoggerApp**.

Lúc này, Visual Studio .NET IDE sẽ tạo một khung sườn đối với đoạn mã của ứng dụng (thí dụ 1-30) , đồng thời cho hiện lên một biểu mẫu, mặc nhiên mang tên Form1.

### ***Thí dụ 1-30: Khung sườn đoạn mã ứng dụng CarLoggerApp***

\*\*\*\*\*

```
using System;
using System.Drawing;
using System.Collections;
using System.ComponentModel;
using System.Windows.Forms;
using System.Data;

namespace CarLoggerApp
{
    public class Form1: System.Windows.Forms.Form
    {
        private System.ComponentModel.IContainer components = null;

        public Form1()
        {
            InitializeComponent();
        }
        . . .
        #region Windows Form Designer generated code

        private void InitializeComponent()
        {
            this.components = new System.ComponentModel.Container();
            this.Size = new System.Drawing.Size(300,300);
            this.Text = "Form1";
        }
        #endregion

        [STAThread]
        static void Main()
        {
            Application.Run(new Form1());
        }
    }
}
```

```

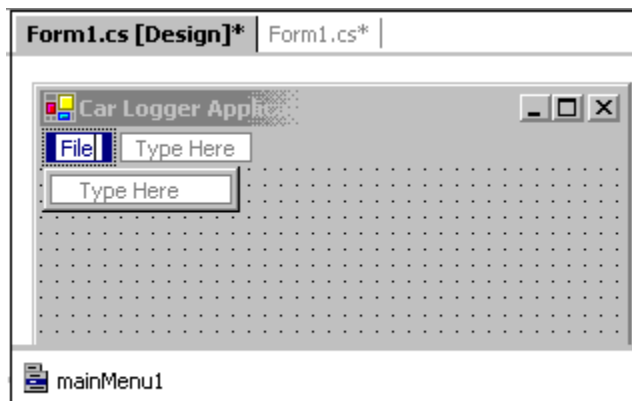
    }
}
}
*****

```

Bạn cho hiệu đính 3 nơi có từ Form1 (mà chúng tôi cho in đậm), thành **mainForm**. Còn trên biểu mẫu Form1, bạn right-click lên biểu mẫu để cho hiện lên trình đơn shortcut, rồi chọn click mục **Properties** để cho hiện lên cửa sổ **Properties**. Trên cửa sổ này, bạn chọn thay đổi tựa đề thông qua thuộc tính **Text**. Thay vì trị mặc nhiên Form1, bạn khở vào “Car Logger Application”. Bạn xem hình 1-23, tựa đề biến thành Car Logger Application. Ngoài ra, thuộc tính Name thay vì mặc nhiên là Form1, bạn khở vào **mainForm**, cho biết là biểu mẫu chính.

### *Thêm trình đơn File với các mục chọn*

Bây giờ bạn tiến hành thêm một trình đơn **File** với các mục chọn như theo bảng 1-17. Muốn thế, bạn lôi mục **MainMenu** trên Toolbox thả lên biểu mẫu. Lúc này trên biểu mẫu hiện lên một khung nhỏ mang dòng chữ “Click Here”, đồng thời ở cuối màn hình hiện lên icon mainMenu1, cho biết là một đối tượng trình đơn chính đang được hình thành. Bạn click lên ô có chữ Click Here, rồi khở vào ‘File’ cho biết là trình đơn File. Lúc này có hai ô với chữ “Click Here” hiện lên (xem hình 1-24), một ô bên phải ô File, và một ô khác phía dưới ô File. Ở đây ta chỉ có một trình đơn File, với 5 mục chọn, do đó ta click ô nằm dưới ô File.



**Hình 01-24: Hình thành một trình đơn**

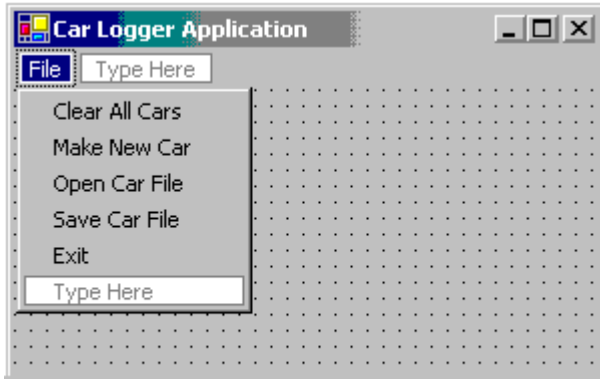
đơn File. Thí dụ, bạn right click lên mục chọn Clear All Cars, rồi chọn click mục Properties để cho hiện lên cửa sổ Properties của mục chọn này. Bạn cho thay đổi tên mặc nhiên của mục chọn này là menuItem2 (trên ô Name) thành menuItemClear giống như trên bảng 1-17 cột Name. Bạn tiếp tục thay đổi thuộc tính Name của các mục chọn khác dựa theo bảng 1-17.

Ta khở vào mục chọn thứ nhất như trên bảng 1-17: **Clear All Cars**. Khi bạn khở xong thì hai ô Click Here lại xuất hiện bên phải và phía dưới mục chọn bạn vừa khở vào. Bạn tiếp tục với các mục chọn còn lại. Cuối cùng, bạn có hình 1-25, là trình đơn **File** với các mục chọn như theo bảng 1-17.

Sau khi đặt đề xong trình đơn File, bạn cho thay đổi các thuộc tính các mục chọn trình

## Cho đặt để một DataGrid

Bây giờ, bạn dùng Toolbox, lôi một DataGrid thả lên biểu mẫu, rồi chỉnh kích thước chiếm trọn biểu mẫu. Bạn right-click lên DataGrid, rồi chọn click lên mục Properties để cho hiện lên cửa sổ Properties. Bạn thay đổi thuộc tính (Name) thành **carDataGrid**, thuộc tính **ReadOnly** về true (ngăn không cho người sử dụng hiệu đính các ô trong hàng dữ liệu), và thuộc tính **CaptionText** thành The auto inventory.



**Hình 01-25: Trình đơn File hoàn tất**

## Tạo lớp Car

Bây giờ ta định nghĩa một lớp mang tên Car với attribute [Serializable] nằm trong namespace CarLoggerApp, như sau:

```
namespace CarLoggerApp
{
    [Serializable]
    public class Car
    { // Cho ra public do dễ truy cập
        public string petName, make, color;

        public Car(string petName, string make, string color)
        { this.petName = petName;
          this.color = color;
          this.make = make;
        }
    }
    ...
}
```

Đây là lớp tượng trưng cho một hàng dữ liệu duy nhất trên DataGrid, nhưng cũng là một item trong object graph được serialized.

## Hiệu đính lớp mainForm

Tiếp theo, lớp mainForm (biểu mẫu chính) duy trì một kiểu dữ liệu ArrayList, private, dùng trữ mỗi qui chiếu về đối tượng Car. Hàm constructor của mainForm sẽ

thêm vài đối tượng Car mặc nhiên để cho hiện lên khi nạp data grid lên biểu mẫu. Một khi các đối tượng Car được thêm vào collection, thì một hàm hỗ trợ UpdateGrid() sẽ cho hiển thị nội dung data grid. Ta có đoạn mã lớp mainForm được hiệu đính như sau. Các hàng ghi đậm là phần hiệu đính mới thêm vào:

```
public class MainForm: System.Windows.Forms.Form
{ // Danh sách dùng cho object serialization
    private ArrayList arTheCars = null;
    public MainForm()
    { InitializeComponent();
      CenterToScreen();

      // Thêm vài chiếc xe
      arTheCars = new ArrayList();
      arTheCars.Add(new Car("Siddhartha", "BMW", "Silver"));
      arTheCars.Add(new Car("Chuck", "Caravan",
                           "Pea Soup Green"));
      arTheCars.Add(new Car("Fred", "Audi TT", "Red"));

      // Cho hiển thị nội dung của datagrid
      UpdateGrid();
    }
}
```

### *Hàm UpdateGrid()*

Hàm **UpdateGrid()** chịu trách nhiệm tạo một bảng dữ liệu kiểu System.Data.Table chứa một hàng dữ liệu đối với mỗi đối tượng Car được trữ trong **ArrayList arTheCars**. Một khi bảng dữ liệu đã được điền đầy, thì cho gắn kết với **DataGrid carDataGrid**. Tập sách IV nói về ADO.NET sẽ đề cập chi tiết về căn cứ dữ liệu. Tạm thời các dòng lệnh sử dụng trong hàm hành sự **UpdateGrid()** làm công việc kể trên. Sau đây là đoạn mã của hàm hành sự **UpdateGrid()**. Bạn để ý bảng dữ liệu cần đến namespace System.Data, do đó phải thêm chỉ thị: **using System.Data;** vào đầu dự án.

```
using System.Data;
...
private void UpdateGrid()
{
    if(arTheCars != null)
    { // Tạo một đối tượng DataTable cho mang tên Inventory
      DataTable inventory = new DataTable("Inventory");
      // Tạo các đối tượng DataColumn
      DataColumn make = new DataColumn("Car Make");
      DataColumn petName = new DataColumn("Pet Name");
      DataColumn color = new DataColumn("Car color");
      // Thêm cột vào bảng dữ liệu Inventory
      inventory.Columns.Add(petName);
      inventory.Columns.Add(make);
      inventory.Columns.Add(color);
      // Rảo qua ArrayList để tạo những hàng dữ liệu
      foreach(Car c in arTheCars)
      { // tạo một hàng dữ liệu
```

```

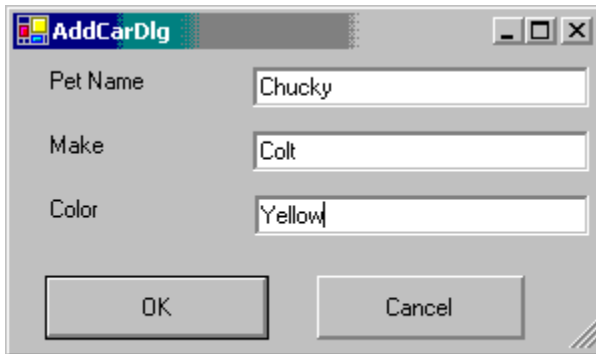
        DataRow newRow;
        newRow = inventory.NewRow();
        newRow["Pet Name"] = c.petName;
        newRow["Car Make"] = c.make;
        newRow["Car Color"] = c.color;
        inventory.Rows.Add(newRow);
    }

    // Gắn kết với datagrid
    carDataGrid.DataSource = inventory;
}
}

```

Theo đoạn mã trên, bạn bắt đầu tạo một đối tượng **Inventory** kiểu dữ liệu **DataTable**. Trong thế giới ADO.NET, một đối tượng **DataTable** là một biểu diễn trong ký ức một bảng dữ liệu. Bạn có thể coi nó như là một thực thể hiện diện một mình.

Một khi bạn đã có một bảng dữ liệu **Inventory**, bạn cần thiết lập một nhóm cột được liệt kê trên bảng dữ liệu. Kiểu dữ liệu **System.Data.DataColumn** tương trưng cho một cột dữ liệu đơn lẻ. Vì kiểu dữ liệu **Car** có 3 vùng mục tin public (**petName**, **make** và **color**) bạn tạo 3 cột dữ liệu và cho chèn vào bảng dữ liệu **Inventory** bằng cách sử dụng thuộc tính **DataTable.Columns**.



**Hình 01-26: Khung đối thoại thêm Car mới**

Tiếp theo, bạn đưa mỗi hàng dữ liệu vào bảng **Inventory**. Bạn còn nhớ **mainForm** duy trì một **ArrayList** chứa một số đối tượng **Car**. Vì **ArrayList** thi công giao diện **IEnumerable** bạn có thể đi tìm mỗi đối tượng **Car** từ **Collection**, đọc các vùng mục tin public hình thành một hàng dữ liệu rồi chèn một **DataRow** vào bảng dữ liệu **Inventory**. Vòng lặp **foreach** lo làm việc này.

Bây giờ, nếu bạn cho **Build** rồi cho chạy thử, bạn sẽ thấy **DataGrid** sẽ điền đầy những chiếc xe mặc nhiên, như theo hình 1-23.

### ***Thi công phần Thêm một đối tượng Car mới***

Dự án **CarLoggerApp** định nghĩa thêm một khung đối thoại modal thuộc lớp **AddCarDlg**. Hình 1-26. Muốn tạo khung đối thoại này, trên **Solution Explorer** của dự án **CarLoggerApp**, bạn right-click lên nhánh **CarLggerApp** để cho hiện lên trình đơn shortcut, rồi bạn ra lệnh **Add | Add Windows Forms...** để cho hiện lên khung đối thoại **Add New Item**. Trên khung đối thoại này, bạn chọn **Windows Form** trên khung

**Template**, rồi kho AddCarDlg.cs vào ô Name rồi ấn <Open>. Lúc này một biểu mẫu AddCarDlg hiện lên. Bạn dùng Toolbox, thêm lên biểu mẫu AddCarDlg 3 ô label, 3 text box (để nhận dữ liệu Pet Name, Make và Color) và hai button (OK và Cancel). Bạn đặt để các thuộc tính của các ô control này như theo hình 1-26. Với khung đối thoại này bạn có thể nhập liệu một đối tượng Car mới. Bạn dùng cửa sổ Properties để thay đổi các thuộc tính của các ô control kể trên. Với các ô text box, thuộc tính Name được đổi thành txtName, txtMake và txtColor. Còn các button thì thuộc tính Name cũng đổi thành btnOK, btnCancel.

Khi bạn tạo xong khung đối thoại nhập liệu AddCarDlg, bạn double-click lên nút <OK>, thì IDE sẽ tạo ra khung sườn hàm hành sự **btnOK\_Click**. Bạn kho tiếp nội dung hàm hành sự **btnOK\_Click** như sau, đồng thời bạn thêm một hàm hỗ trợ SetupOKCancel phát hiện kết quả nút OK hoặc Cancel bị ấn:

```
public class AddCarDlg: System.Windows.Forms.Form
{ // Cho về public để dễ truy xuất
    public Car theCar = null;
    . . .
    protected void btnOK_Click(object sender, System.EventArgs e)
    { // Cấu hình một đối tượng Car mới khi nút OK bị ấn xuống
        theCar = new Car(txtName.Text, txtMake.Text, txtColor.Text);
        this.Close();
    }

    public void SetupOKCancel()
    { btnOK.DialogResult = DialogResult.OK;
      btnCancel.DialogResult = DialogResult.Cancel;
      AcceptButton = btnOK;
      CancelButton = btnCancel;
    }
}
```

Bạn thấy là nút OK được gán thuộc tính DialogResult.OK còn nút Cancel thì được gán thuộc tính DialogResult.Cancel. Hai thuộc tính AcceptButton và CancelButton là những thuộc tính của biểu mẫu AddCarDlg.

Biểu mẫu MainForm cho hiển thị khung đối thoại AddCarDlg khi người sử dụng ra lệnh **File | Make New Car**. Sau đây là đoạn mã khi bạn ra lệnh trên.

```
protected void menuItemNewCar_Click(object sender,
                                     System.EventArgs e)
{
    // Cho hiển thị khung đối thoại và kiểm tra nút OK
    AddCarDlg d = new AddCarDlg();
    d.SetupOKCancel();
    d.ShowDialog();

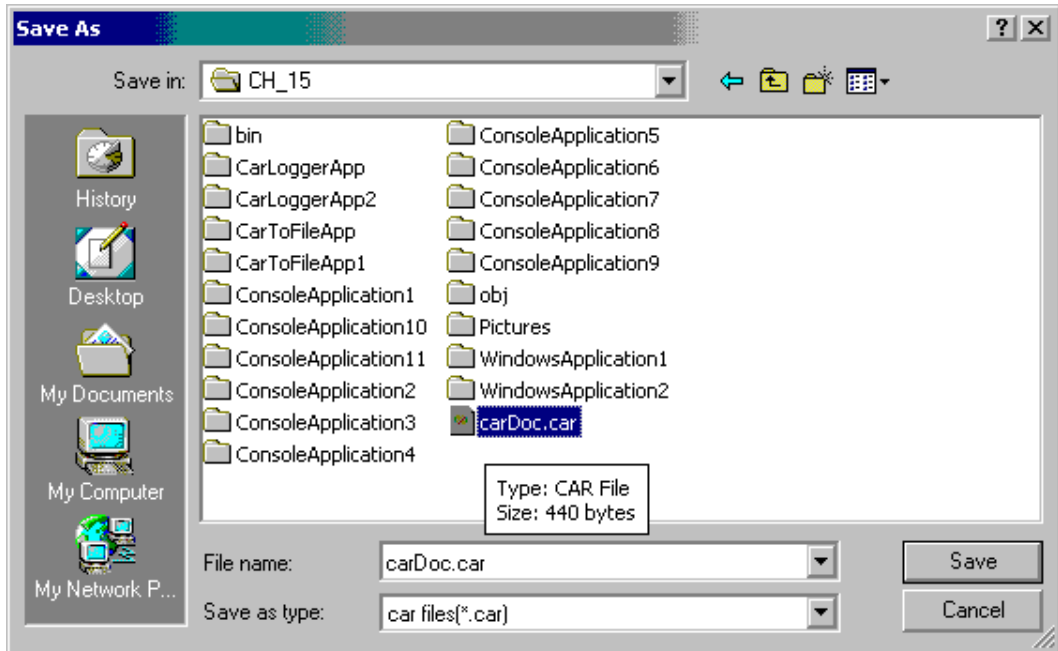
    if(d.DialogResult == DialogResult.OK)
    { // Thêm một xe mới vào arraylist
      arTheCars.Add(d.theCar);
    }
}
```

```

        UpdateGrid();
    }
}

```

Trong đoạn mã này bạn cho hiện lên khung đối thoại AddCarDlg như là một khung đối thoại modal, và nếu nút OK bị click bạn cho hình thành một hàng dữ liệu lên ArrayList rồi cho refresh DataGridView.



**Hình 01-27: Khung đối thoại File Save chuẩn**

### *Cho Serialization đối tượng Car*

Đoạn mã xử lý các tình huống **File | Save Car File** và **File | Open Car File** sẽ không có vấn đề gì. Khi người sử dụng muốn cho cất trữ tồn kho hiện hành, bạn tạo một tập tin mới và sử dụng Binary Formatter để serialize các đối tượng. Tuy nhiên, để cho hấp dẫn, người sử dụng có thể thiết lập tên và lối tìm về tập tin này sử dụng một **System.Windows. Forms.SaveFileDialog**. Kiểu dữ liệu này sẽ cho hiện lên một khung đối thoại chuẩn Save As như theo hình 1-27:

Sau đây là đoạn mã thụ lý tình huống **File | Save Car File**:

```

protected void menuItemSave_Click(object sender, System.EventArgs e)
{
    // Cấu hình một khung đối thoại Save

```



```

SaveFileDialog mySaveFileDialog = new SaveFileDialog();
mySaveFileDialog.InitialDirectory = ".";
mySaveFileDialog.Filter = "car files (*.car) | *.car |
                          All files (*.*) | *.*";
mySaveFileDialog.FilterIndex = 1;
mySaveFileDialog.RestoreDirectory = true;
mySaveFileDialog.FileName = "carDoc";

// Có một tập tin chưa?
if(mySaveFileDialog.ShowDialog() == DialogResult.OK)
{
    Stream myStream = null;
    if((myStream = mySaveFileDialog.OpenFile()) != null)
    {
        // Save các xe
        BinaryFormatter myBinaryFormat = new BinaryFormatter();
        myBinaryFormat.Serialize(myStream, arTheCars);
        myStream.Close();
    }
}
}

```

Bạn để ý thuộc tính **Filter** của đối tượng **SaveFileDialog**. Thuộc tính này nhận chuỗi phân cách bởi OR tượng trưng cho những dòng văn bản được dùng trong các combo box kéo xuống “File Name” và “Save As Type”.

Khi sử dụng **BinaryFormatter** trong hàm hành sự này, bạn phải thêm các chỉ thị using sau đây vào đầu dự án:

```

using System.Runtime.Serialization;
using System.Runtime.Serialization.Formatters.Binary;

```

Bạn để ý là **OpenFile()** của lớp **SaveFileDialog** trả về một **Stream** tượng trưng cho tập tin được chọn ra bởi người sử dụng.

Đoạn mã thụ lý tình huống **File | Open Car File** cũng tương tự như với **File | Save Car File**. Lần này bạn tạo một đối tượng của lớp **System.Windows.Forms.OpenFileDialog**, cho cấu hình một cách thích ứng rồi nhận một qui chiếu về **Stream** dựa theo tập tin được chọn ra. Tiếp theo bạn dump nội dung của **ArrayList** rồi đọc vào object graph sử dụng **BinaryFormatter.Deserialize()**. Sau đây là đoạn mã thụ lý tình huống **File | Open Car File**:

```

protected void menuItemOpen_Click(object sender, EventArgs e)
{
    // Cấu hình một khung đối thoại Open
    OpenFileDialog myOpenFileDialog = new OpenFileDialog();
    myOpenFileDialog.InitialDirectory = ".";
    myOpenFileDialog.Filter = "car files (*.car) | *.car |
                              All files (*.*) | *.*";
    myOpenFileDialog.FilterIndex = 1;
    myOpenFileDialog.RestoreDirectory = true;

    // Có một tập tin chưa?

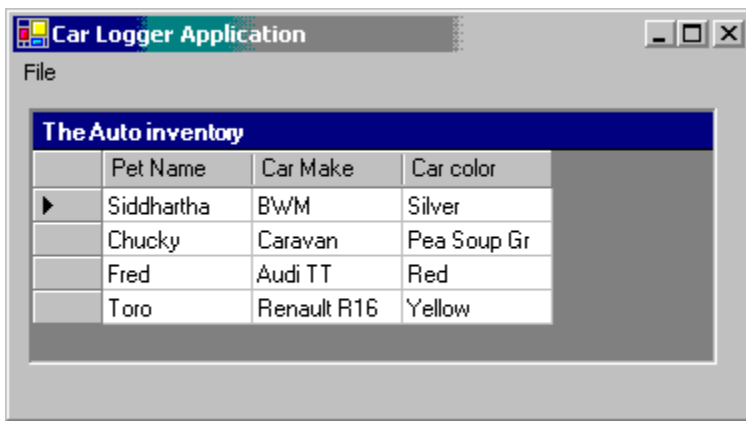
```

```

if(myOpenFileDialog.ShowDialog() == DialogResult.OK)
{
    // Xóa trống arraylist hiện hành
    arTheCars.Clear();

    Stream myStream = null;
    if((myStream = myOpenFileDialog.OpenFile()) != null)
    {
        // Đọc dữ liệu các đối tượng Car
        BinaryFormatter myBinaryFormat = new BinaryFormatter();
        arTheCars = (ArrayList)myBinaryFormat.Deserialize(myStream);
        myStream.Close();
        UpdateGrid();
    }
}
}

```



Tới đây, chỉ còn lại hai mục chọn trình đơn chót là **File | Clear All Cars** và **File | Exit**. Các đoạn mã thụ lý các tình huống này sẽ như sau và chả có gì tối tăm để giải thích thêm:

**Hình 01-28: Thêm một Car mới**

```

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

protected void menuItemClear_Click(object sender, System.EventArgs e)
{
    arTheCars.Clear();
    UpdateGrid();
}

```

Bạn có thể Build rồi cho chạy thử. Bạn thêm một mẫu tin mới, bạn thấy là nó được thêm vào như theo hình 1-28. Bạn có thể thử các mục chọn khác thấy là chạy tốt.

## Chương 2

# Xây dựng một ứng dụng Windows

Khi thiết kế những ứng dụng đòi hỏi có một giao diện đồ họa tương tác với người sử dụng (graphical user interface – GUI), .NET cung cấp cho bạn hai lựa chọn: **Windows Forms** và **Web Forms**. Cả hai cho phép bạn xây dựng những ứng dụng cổ điển chạy trên máy để bàn (desktop) cũng như những ứng dụng phát tán (distributed application) giải quyết những vấn đề sản xuất kinh doanh của công ty trong thời đại Internet ngày nay.

Nếu khách hàng yêu cầu bạn triển khai một ứng dụng đòi hỏi xử lý nhanh khối lượng dữ liệu khá lớn, chẳng hạn trong kế toán, tồn kho vật tư hoặc hoá đơn bán hàng, thì lẽ dĩ nhiên bạn chọn **Windows Forms**. Đây bao gồm những ứng dụng desktop cổ điển Win32, mà người ta thường triển khai sử dụng đến các ngôn ngữ Visual Basic hoặc Visual C++.

Còn ngược lại, nếu công ty của bạn muốn tiến vào lĩnh vực thương mại điện tử (e-commerce) thông qua Web site trên Internet thì bạn phải triển khai ứng dụng sử dụng Web Forms. ASP .NET Web Forms được dùng để tạo những ứng dụng theo đầy giao diện GUI chủ yếu là một browser<sup>3</sup>. Phần lớn các kiểu dữ liệu của **Web Forms** nằm trên các namespace **System.Web.UI** và **System.Web.UI.WebControls**. Sử dụng các kiểu dữ liệu này, bạn có thể xây dựng những ứng dụng độc lập với browser dựa trên những chuẩn khác nhau (HTML, HTTP, v.v.). Trong chương này chúng tôi không đề cập đến ASP.NET, mà sẽ giải thích chi tiết trong tập V của bộ sách này, khi đề cập đến “Lập trình ASP.NET”.

Bạn cũng nên để ý là mặc dù **Windows Forms** và **Web Forms** cùng chia sẻ một vài kiểu dữ liệu mang tên giống nhau (chẳng hạn Button, CheckBox, v.v..) nhưng chúng không chia sẻ cùng thiết kế thi công và không thể được đối xử giống nhau. Tuy nhiên, một khi bạn đã hiểu sâu namespace **Windows Forms**, thì bạn sẽ thấy việc học sử dụng **Web Forms** cũng chẳng khó khăn gì.

Chương này tập trung vào việc xây dựng những ứng dụng cổ điển Win32 sử dụng namespace **System.Windows.Forms**. Một ứng dụng Windows sử dụng đến **Windows**

---

<sup>3</sup> Người ta dịch “browser” là “trình duyệt”. Theo thiên ý, đây là dịch sai. Thật ra từ “browse” có nghĩa là cho “rảo xem” giống như bạn rảo quanh chợ xem hàng.

**Forms** thường được xây dựng xung quanh một khuôn giả Windows, nên nó có thể truy xuất các nguồn lực hệ thống trên máy khách hàng, bao gồm các tập tin tại chỗ, Window registry, máy in, v.v.. Ngoài ra, **Windows Forms** tận dụng được các lớp .NET GDI+ để tạo một giao diện đồ họa rất bắt mắt, một đòi hỏi thường thấy trong các ứng dụng game.

## 2.1 Tổng quan về namespace Windows.Forms

Namespace **System.Windows.Forms** chứa vô số lớp giúp bạn tạo những ứng dụng Windows tận dụng những chức năng giao diện người sử dụng phong phú có sẵn trên hệ điều hành Windows. Cũng như bất cứ namespace nào, **System.Windows.Forms** thường gồm một số lớp, cấu trúc, giao diện, delegate và enumeration. Trong những chương kế tiếp bạn sẽ làm quen với vô số lớp thuộc namespace này.

Các lớp trong namespace có thể chia làm những loại sau đây: **Control**, **User Control**, **Form**, **Controls**, **Components** và **Common Dialog Box**:

- **Control, User Control và Form**

Phần lớn các lớp thuộc namespace **System.Windows.Forms** thường được dẫn xuất từ lớp **Control**. Lớp **Control** cung cấp những chức năng cơ bản đối với tất cả các ô control được hiển thị trên một đối tượng **Form** (biểu mẫu). Lớp **Form** tượng trưng cho một cửa sổ nằm trong một ứng dụng, bao gồm khung đối thoại, cửa sổ modeless, và Multiple Document Interface (MDI). Muốn tạo một ô control “cây nhà lá vườn” (custom control) do kết hợp các ô control khác, bạn có thể dùng lớp **UserControl**.

- **Controls**

Namespace **System.Windows.Forms** cung cấp vô số ô control khác nhau cho phép bạn tạo một giao diện người sử dụng phong phú. Một vài ô control được sử dụng vào việc nhập liệu (data entry), chẳng hạn **TextBox** và **ComboBox**. Còn một số khác thì lại cho hiển thị dữ liệu của ứng dụng, chẳng hạn **Label** và **ListView**. Một số ô control khác lo triệu gọi hàm, chẳng hạn **Button** và **ToolBar**. Ngoài ra, ô control **PropertyGrid** có thể được sử dụng để tạo trình thiết kế Windows Forms riêng của bạn để cho hiển thị những thuộc tính của các ô control có thể nhìn thấy được vào lúc thiết kế.

- **Components**

Ngoài ra, namespace **System.Windows.Forms** còn cung cấp những lớp khác không được dẫn xuất từ lớp **Control**, nhưng vẫn cung cấp những chức năng nhìn thấy được đối với một ứng dụng Windows. Một vài lớp, chẳng hạn **ToolTip** và **ErrorProvider**, nói rộng khả năng hoặc cung cấp thông tin cho người sử dụng. Một số lớp khác, chẳng hạn **Menu**, **MenuItem** và **ContextMenu**, cung cấp khả năng hiển thị những

trình đơn cho phép người sử dụng triệu gọi lệnh trong một ứng dụng. Các lớp **Help** và **HelpProvider** cho phép bạn hiển thị thông tin help cho người sử dụng ứng dụng của bạn.

- **Common Dialog Boxes**

Windows cung cấp cho bạn vô số khung đối thoại<sup>4</sup>, có thể được dùng đem lại cho ứng dụng một giao diện GUI nhất quán khi thi hành những công tác thông dụng như mở một tập tin hoặc cất trữ một tập tin chẳng hạn, hoặc thay đổi màu sắc, phông chữ hoặc in ra văn bản v.v.. Các lớp **OpenFileDialog** và **SaveFileDialog** cho phép bạn hiển thị một khung đối thoại để người sử dụng có thể rà soát hoặc gõ vào tên một tập tin cần mở hoặc cho cất trữ. Lớp **FontDialog** cho hiển thị một khung đối thoại để thay đổi những phần tử của đối tượng **Font** được sử dụng trong ứng dụng. Các lớp **PageSetupDialog**, **PrintPreviewDialog** và **PrintDialog** sẽ cho hiển thị các khung đối thoại cho phép người sử dụng điều khiển những khía cạnh khác nhau trong việc in tài liệu. Ngoài ra, namespace **System.Windows.Forms** còn cung cấp lớp **MessageBox** dùng hiển thị một thông điệp hoặc nhận dữ liệu từ người sử dụng.

Ngoài ra, namespace **System.Windows.Forms** còn cung cấp một số lớp hỗ trợ các lớp vừa kể trên. Thí dụ các lớp hỗ trợ là enumeration, event argument và delegate thường được dùng bởi event trong lòng các ô control và component.

Bảng 2-1 sau đây liệt kê một vài lớp cốt lõi của namespace **System.Windows.Forms**. Phải vài chục trang mới hết.

**Bảng 2-1: Các lớp cốt lõi của System.Windows.Forms**

Tên lớp	Mô tả
<b>Application</b>	Lớp này tượng trưng cho một ứng dụng Windows Forms. Nó cung cấp những hàm hành sự và thuộc tính lo quản lý một ứng dụng, chẳng hạn hàm hành sự lo khởi động hoặc ngưng một ứng dụng, lo xử lý những thông điệp Windows, cũng như những thuộc tính lo nhận thông tin liên quan đến ứng dụng. Lớp này không thể được dẫn xuất.
<b>ButtonBase, Button, CheckBox, ComboBox, DataGrid, GroupBox, ListBox, LinkLabel, PictureBox</b>	Các lớp này (ngoài một số khá lớn) tượng trưng cho những kiểu dữ liệu tương ứng với những GUI widgets (lục lăng lục chốt). Bạn có thể xem chi tiết các lớp này ở chương 6, Tập III.
<b>Form</b>	Lớp này tượng trưng cho cửa sổ chính (hoặc một khung đối thoại) của một ứng dụng Windows Forms.

---

<sup>4</sup> Có người dịch là “hộp thoại”.

<b>ColorDialog, FileDialog, FontDialog, PrintPreviewDialog</b>	Như bạn có thể thấy, Windows.Forms định nghĩa một số lớp được “đóng hộp”(canned). Nếu bạn không vừa lòng, bạn có quyền tạo những khung đối thoại “cây nhà lá vườn” theo ý muốn riêng của bạn.
<b>Menu, MainMenu, MenuItem, ContextMenu</b>	Các lớp này cho phép bạn tạo những hệ thống trình đơn chính thống cũng như trình đơn shortcut.
<b>Clipboard, Help, Timer, Screen, ToolTip, Cursors</b>	Đây là những lớp tiện ích giúp tương tác với GUI.
<b>StatusBar, Splitter, Toolbar, ScrollBar</b>	Đây là những lớp khác nhau dùng tô điểm một Form với những ô control con-cái thông dụng.

## 2.2 Tương tác với các lớp Windows Forms

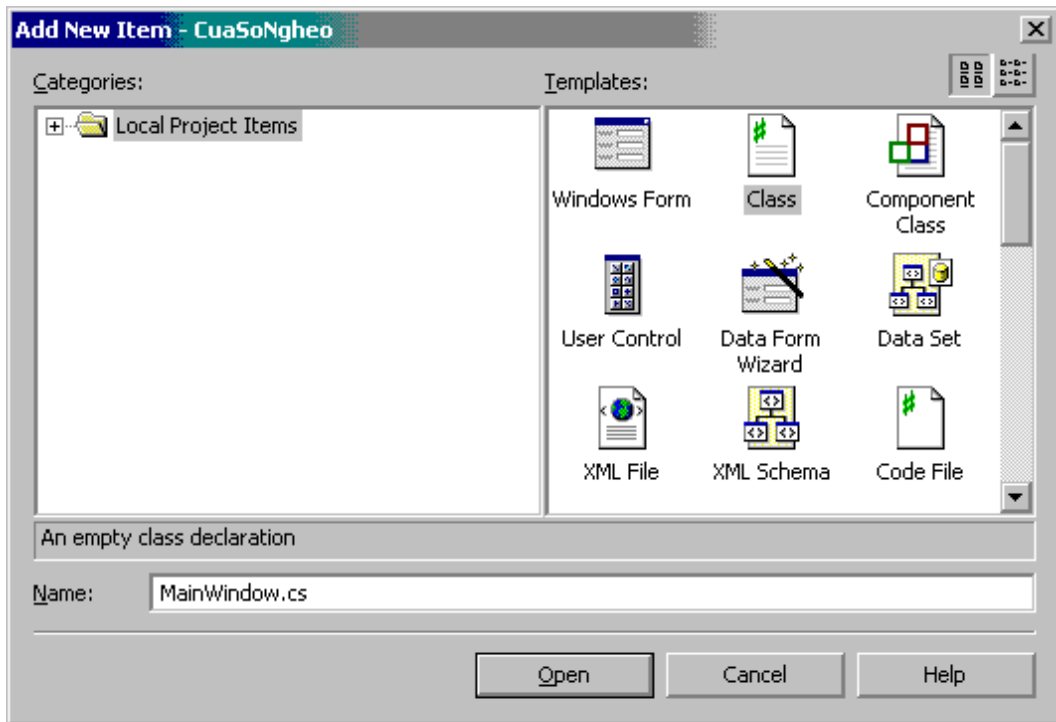
Khi bạn viết những ứng dụng **Windows Forms**, có thể bạn chọn viết tất cả các đoạn mã bằng tay, sử dụng Notepad chẳng hạn, tạo một tập tin **.cs** rồi triệu gọi trình biên dịch C# **csc.exe** với flag **/target:winexe** từ command line. Mất thời gian để viết những ứng dụng Windows Forms bằng tay không phải là cách hay nhất để có kinh nghiệm. Tuy nhiên đây cũng là cách bạn hiểu những đoạn mã kết sinh bởi các GUI wizard khác nhau.

Một lựa chọn khác là tạo những dự án **Windows Forms** sử dụng đến môi trường Visual Studio .NET IDE. IDE cung cấp cho bạn vô số wizard, template và công cụ cho phép bạn làm việc thoải mái với các lớp **Windows Forms**.

Vấn đề đối với những wizard là bạn không hiểu gì hết đối với những đoạn mã được kết sinh giùm bạn, nghĩa là bạn không nắm chắc công nghệ lập trình nằm ở đằng sau. Do đó, ta nên bắt đầu bằng cách viết những chương trình Windows Forms thô thiển nhất (trộn bộ với trình đơn, thanh tình trạng và thanh công cụ) và minh họa việc sử dụng những wizard mà Visual Studio .NET cung cấp khi thấy cần thiết.

### 2.2.1 Tạo một Windows Form đơn giản

Để bắt đầu tìm hiểu việc lập trình Windows Forms, ta thử xây dựng một cửa sổ chính đơn giản bằng tay. Ta bắt đầu tạo một mặt bằng làm việc mới cho dự án C# rỗng, cho đặt tên dự án là “CuaSoNgheo” sử dụng Visual Studio .NET IDE. Bạn ra lệnh **File | New Project** để cho hiện lên khung đối thoại **New Project**. Rồi bạn chọn **Visual C# project**, và **Empty Project**, và đặt tên cho dự án và kho vào tên thư mục. Tiếp theo, ta chèn vào một định nghĩa lớp C# mới bằng cách ra lệnh **Project | Add Class**. Lớp này cho mang tên **MainWindow** (hình 2-1).



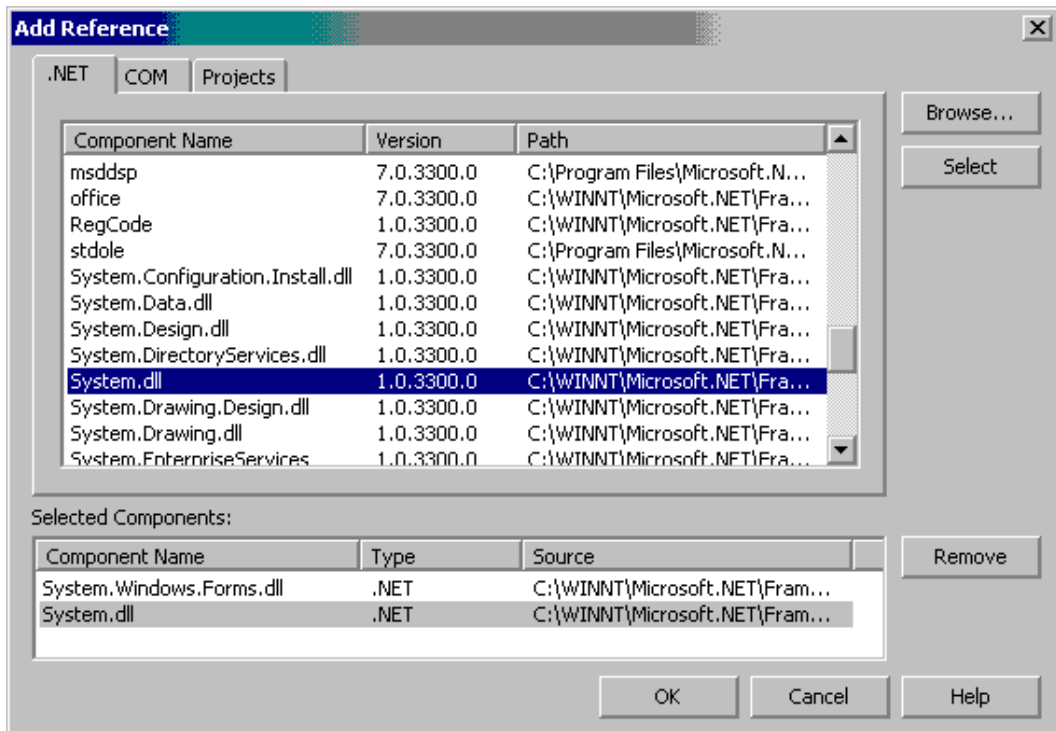
Hình 2-1: Cửa sổ Add New Item. Thêm một lớp C#

Khi bạn xây dựng một cửa sổ chính bằng tay, ít nhất bạn cần đến các lớp **Form** và **Application**, cả hai nằm trên namespace **System.Windows.Forms.dll**. Một ứng dụng Windows Forms cũng cần qui chiếu **System.dll**. Nói tóm lại bạn phải thêm các reference này vào assembly, bằng cách ra lệnh **Project | Add Reference...** Xem hình 2-2. Khi cửa sổ **Add Reference** hiện lên, bạn lần lượt cho ngồi sáng **System.dll** và **System.Windows.Forms.dll** rồi ấn nút <Select>, cuối cùng ấn <OK>. Lúc này các namespace qui chiếu được ghi lên cửa sổ Solution Explorer.

Bạn cho biên dịch, bằng cách ấn phím <F5>. Cửa sổ mã nguồn của **MainWindow.cs** hiện lên có vài chỗ để bạn điền vào.

## 2.2.2 Tạo bằng tay một cửa sổ chính

Trong thế giới Windows Forms, đối tượng **Form** tượng trưng cho bất cứ cửa sổ nào trong ứng dụng; bao gồm cửa sổ chính cao nhất trong một ứng dụng SDI (Single Document Interface), các khung đối thoại modal cũng như modeless, kể cả những cửa sổ cha-mẹ và con-cái trong một ứng dụng MDI (Multiple Document Interface). Khi bạn quan tâm đến việc tạo một cửa sổ chính, bạn phải tiến hành hai bước bắt buộc.



Hình 2-02: Thêm qui chiếu thông qua khung đối thoại Add Reference

- Cho dẫn xuất một lớp custom mới từ **System.Windows.Forms.Form**. Lớp biểu mẫu bạn dẫn xuất từ Form, được gọi là custom form (biểu mẫu cây nhà lá vườn).
- Cấu hình hoá hàm **Main()** của ứng dụng cho triệu gọi hàm **Application.Run()**, trao một thể hiện của lớp mới được dẫn xuất từ Form như là đối mục của **Run()**.

Với những bước kể trên, bạn có thể nhậ tu phần định nghĩa lớp MainWindow ban đầu, như sau:

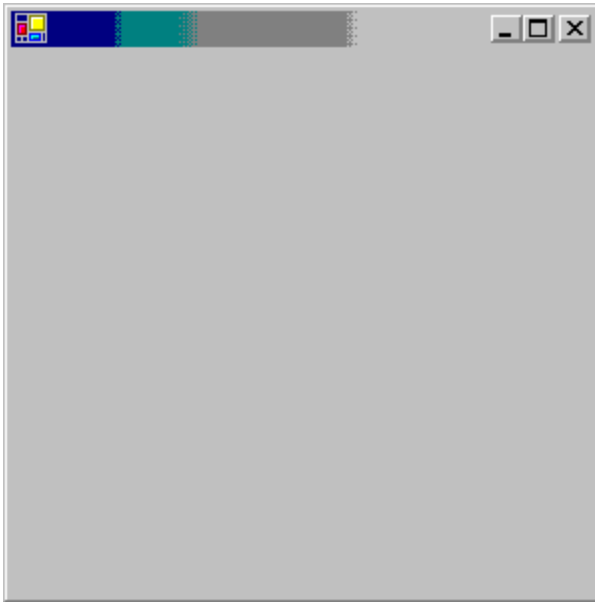
```
namespace CuaSoNgheo
{
    using System;
    using System.Windows.Forms;

    public class MainWindow: Form
    {
        public MainWindow(){}
        // cho chạy ứng dụng này
        public static int Main(string[] args)
        {
            Application.Run(new MainWindow());
            return 0;
        }
    }
}
```



```
}
```

Phần thêm vào được in đậm. Bạn ấn phím <F5> cho chạy chương trình. Kết quả là hình 2-3.



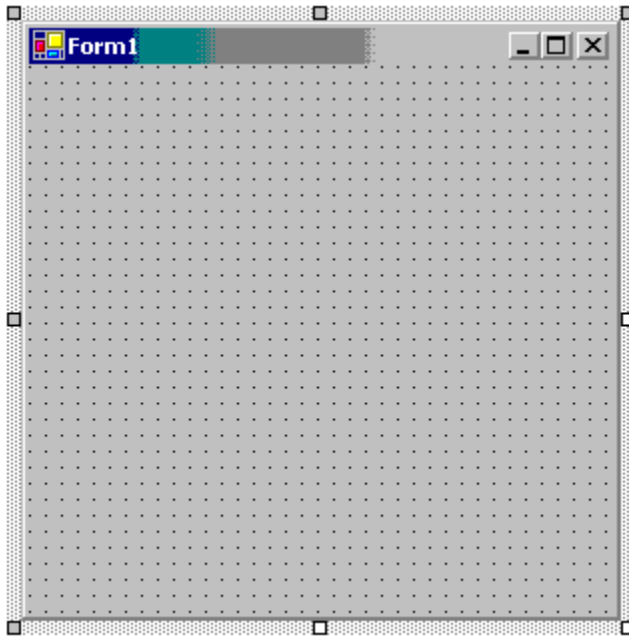
Hình 2-03: Một biểu mẫu cơ bản

Bạn thấy kết quả giống như đưa trẻ sơ sinh trần truồng chả có chi cả ngoài mấy cái nút Minimize, Maximize, Close, và một icon. Khi chương trình được khởi động bạn để ý là phía hậu trường có màn hình command line. Lý do là vì ta chưa cấu hình những đặt để xây dựng (build setting) đối với việc kết sinh ứng dụng Windows EXE. Để khắc phục, trên cửa sổ Solution Explorer, bạn right-click lên **CuaSoNgheo** để cho hiện lên trình đơn shortcut, rồi bạn chọn mục **Property**. Khung đối thoại **Property Pages** hiện lên, bạn cho bung mắt gút “**Common Properties | General**” rồi cho đổi **Output type** thành **Windows Application**. Bạn cho biên dịch và cho chạy lại thì lúc này màn hình command line không còn nữa.

Như vậy, tới đây bạn có được một cửa sổ chính có thể teo nhỏ, phình lớn và đóng lại được. Ngoài ra, cửa sổ còn có một icon nằm ở bên trái, cho phép bạn khởi động. So với những ai đã từng lập trình trên Visual C++ hoặc với Win32 API, bạn thấy là bạn sướng hơn người ta nhiều. Tuy nhiên, cửa sổ của ta hiện còn vô tích sự.

### 2.2.3 Tạo một dự án Windows Form trên Visual Studio .NET

Bây giờ bạn tạo một dự án ứng dụng Windows, bằng cách ra lệnh **File | New Project**, rồi chọn **Visual C# Projects** và **Windows Application**, cho đặt tên **VSWinApp**, rồi ấn <OK>.



Hình 2-04: Design time template

Khi bạn ấn <OK>, bạn sẽ thấy bạn tự động nhận được một lớp được dẫn xuất từ **System.Windows.Forms.Form**, với hàm **Main()** thích ứng và những qui chiếu về những assembly cần thiết kể cả vài assembly hỗ trợ khác. Ngoài ra, bạn cũng nhận được một khuôn mẫu thiết kế (design time template, xem hình 2-4), một loại bản vẽ, mà bạn có thể dùng để hình thành giao diện GUI. Vào lúc thiết kế, khi bạn bổ sung gì đó, một ô control chẳng hạn, lên khung thiết kế này, xem như *gián tiếp* bạn thêm những đoạn mã vào lớp được dẫn xuất từ Form (ở đây là lớp Form1).

Trên cửa sổ Solution Explorer, nếu bạn right-click lên **Form1.cs** chẳng hạn, một trình đơn shortcut hiện lên, cho phép bạn chọn xem đoạn mã C# được kết sinh (click mục **View Code**) hoặc “bản vẽ” (click mục **View Designer**). Tuy nhiên, bạn cũng có thể double-click lên “bản vẽ” thì sẽ hiện lên đoạn mã C# được kết sinh, nhưng việc này có thể sẽ viết ra một trình thụ lý tình huống (event handler) mà bạn không mong muốn, một tình huống **Load** của Form. Sau đây là đoạn mã kết sinh C#, sau khi cho lược bỏ những hàng chú giải XML:

```
using System;
using System.Drawing;
using System.Collections;
using System.ComponentModel;
using System.Windows.Forms;
using System.Data;

namespace VSWinApp
{
    public class Form1: System.Windows.Forms.Form
    {
        private System.ComponentModel.Container components = null;

        public Form1()
        {
            InitializeComponent();
        }

        protected override void Dispose(bool disposing)
        {
            if(disposing)
```

```

        {
            if (components != null)
            {
                components.Dispose();
            }
        }
        base.Dispose(disposing);
    }

    #region Windows Form Designer generated code
    private void InitializeComponent()
    {
        this.components = new System.ComponentModel.Container();
        this.Size = new System.Drawing.Size(300, 300);
        this.Text = "Form1";
    }
    #endregion

    static void Main()
    {
        Application.Run(new Form1());
    }
}

```

Như bạn có thể thấy đoạn mã này cũng giống như đoạn mã của thí dụ **CuaSoNgheo**. Lớp vẫn được dẫn xuất từ **System.Windows.Forms.Form**, và hàm **Main()** vẫn triệu gọi hàm **Application.Run()**.

Một thay đổi chủ yếu là một hàm hành sự mới được thêm vào, hàm **InitializeComponents()**. Hàm này được bao bởi cặp chỉ thị tiền xử lý (**#region** - **#endregion**), với chú giải “Windows Form Designer generated code” cho biết là đoạn mã kết sinh của Windows Form Designer:

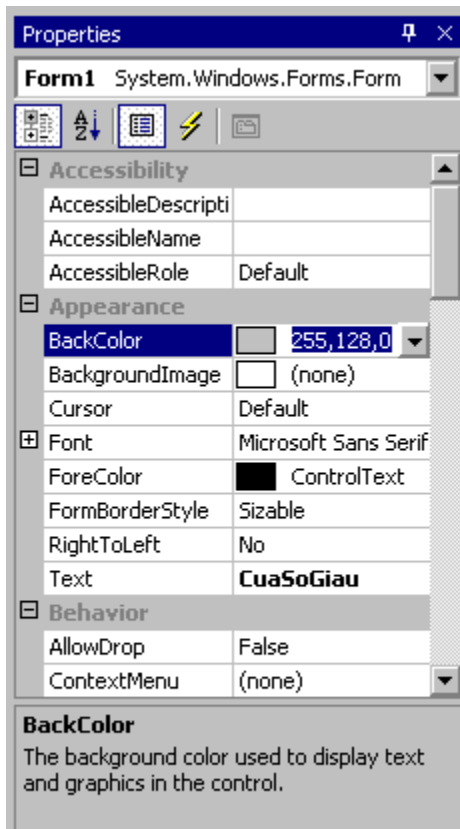
```

#region Windows Form Designer generated code
private void InitializeComponent()
{
    this.components = new System.ComponentModel.Container();
    this.Size = new System.Drawing.Size(300, 300);
    this.Text = "Form1";
}
#endregion

```

Chắc bạn còn nhớ, khối lệnh nào nằm trong cặp chỉ thị này có thể cho teo lại, thay thế bởi một dòng chú giải với dấu (+) nằm ở bên trái.

Hàm **InitializeComponents()** sẽ tự động được nhậ tu bởi form designer để phản ảnh thay đổi mà bạn đã thực hiện trên biểu mẫu cũng như trên các ô control thông qua cửa sổ **Properties**, hình 2-5.



Hình 2-05: Cửa sổ Properties

Thí dụ, nếu bạn sử dụng cửa sổ **Properties** để thay đổi thuộc tính **Text** (để đặt tựa đề cho biểu mẫu) hoặc thuộc tính **BackColor** (để thay đổi màu hậu trường), bạn sẽ thấy là hàm **InitializeComponent()** cũng bị thay đổi theo như sau:

```
#region Windows Form Designer generated
code

private void InitializeComponent()
{
    this.AutoScaleBaseSize = new
        System.Drawing.Size(5, 13);
    this.BackColor =
        System.Drawing.Color.FromArgb(
            ((System.Byte)(255)),
            ((System.Byte)(128)),
            ((System.Byte)(0)));
    this.ClientSize = new
        System.Drawing.Size(292, 273);
    this.Name = "Form1";
    this.Text = "CuaSoGiau";
}
#endregion
```

Lớp dẫn xuất **Form1** triệu gọi hàm **InitializeComponent()** ngay trong phạm vi của hàm constructor:

```
public Form1() // hàm constructor
{
    InitializeComponent();
}
```

Điểm cuối cùng là hàm hành sự bị phủ quyết **Dispose()**. Hàm này sẽ tự động được triệu gọi khi biểu mẫu sắp sửa bị hủy. Và đây là nơi an toàn bạn có thể hủy hoặc giao trả bất cứ nguồn lực chiếm dụng, nay không dùng nữa. Sau đây là đoạn mã của hàm này:

```
protected override void Dispose(bool disposing)
{
    if(disposing)
    {
        if (components != null)
        {
            components.Dispose();
        }
    }
    base.Dispose(disposing);
}
```

Bây giờ bạn đã biết hai cách tiếp cận trong vấn đề tạo một ứng dụng Windows Form. Ta nên dành chút thời gian nghiên cứu sâu hơn các chức năng của lớp **Application**.

## 2.3 Tìm hiểu lớp **System.Windows.Forms.Application**

Lớp **Application** là một lớp cấp thấp bao gồm những thành viên cho phép bạn kiểm soát những hành vi khác nhau của một ứng dụng Windows Forms. Lớp **Application** cung cấp những hàm hành sự và thuộc tính static dùng trong việc quản lý một ứng dụng.

Lớp **Application** có những hàm hành sự để khởi động và ngưng hoạt động của ứng dụng hoặc của các mạch trình (thread), cũng như xử lý các thông điệp Windows. Bạn triệu gọi hàm **Run()** để khởi động một vòng lặp thông điệp ứng dụng (application message loop) trên mạch trình hiện hành, và có thể cho hiện lên một biểu mẫu. Bạn triệu gọi hàm các hàm **Exit()** hoặc **ExitThread()** để cho ngưng vòng lặp xử lý thông điệp. Bạn triệu gọi hàm **DoEvents()** để xử lý những thông điệp khi chương trình bạn đang ở trong một vòng lặp. Bạn triệu gọi hàm **AddMessageFilter()** để thêm một cái lọc thông điệp (message filter) vào “máy bơm thông điệp ứng dụng (application message pump) để điều khiển việc xử lý các thông điệp. Giao diện **IMessageFilter** cho phép bạn ngưng gây ra một tình huống hoặc thi hành những tác vụ đặc biệt trước khi triệu gọi một hàm thụ lý tình huống (event handler).

Nhìn chung, phần lớn công việc, bạn khởi tương tác trực tiếp với lớp **Application** này. Tuy nhiên, ta thử xem vài hành vi của lớp.

### ***Bảng 2-2: Các hàm hành sự cốt lõi của lớp Application***

<b>AddMessageFilter</b>	Thêm một message filter để điều khiển những thông điệp Windows, khi chúng được dẫn dắt về nơi đến của chúng. Hàm này cho phép ứng dụng của bạn chặn lấy những thông điệp để xử lý trước nếu thấy cần thiết. Khi bạn thêm một message filter, bạn phải khai báo một lớp thi công giao diện <b>IMessageFilter</b> .
<b>DoEvents</b>	Cho xử lý tất cả các thông điệp Windows hiện nằm trong hàng nối đuôi các thông điệp, trong một công tác đòi hỏi thời gian dài (chẳng hạn xây dựng một vòng lặp). Bạn có thể nghĩ <b>DoEvents()</b> như là một hạ sách để mô phỏng những hành vi đa mạch trình.
<b>Exit</b>	Thông báo cho tất cả các “máy bơm thông điệp” (message pumps) phải chấm dứt, và cho đóng lại tất cả các cửa sổ ứng dụng sau khi các thông điệp đã được xử lý xong.
<b>ExitThread</b>	Cho thoát khỏi vòng lặp thông điệp trên mạch trình hiện hành và cho đóng lại tất cả các cửa sổ trên mạch trình.

<b>OleRequired</b>	Khởi gán các thư viện OLE trên mạch trình hiện hành. Xem như là triệu gọi tương đương bằng tay <b>OleInitialize()</b> trên .NET.
<b>RemoveMessageFilter</b>	Cho gỡ bỏ một message filter khỏi message pump của ứng dụng.
<b>Run</b>	<i>Overloaded.</i> Bắt đầu cho chạy một vòng lặp các thông điệp (message loop) của một ứng dụng chuẩn, trên mạch trình hiện hành.

Lớp **Application** cũng định nghĩa một số thuộc tính static phần lớn toàn là read-only. Khi bạn quan sát bảng 2-3 sau đây, bạn nên xem mỗi thuộc tính tương trưng cho một vài đặc tính “cấp ứng dụng” chẳng hạn tên công ty, phiên bản ứng dụng v.v.. Chắc bạn đã quen với một số thuộc tính này.

**Bảng 2-3: Các thuộc tính chủ yếu của lớp Application**

<b>CommonAppDataRegistry</b>	Tìm lấy registry key đối với dữ liệu ứng dụng được chia sẻ sử dụng giữa tất cả các người sử dụng.
<b>CompanyName</b>	Đi lấy tên công ty được gắn liền với ứng dụng.
<b>CurrentCulture</b>	Đi lấy hoặc đặt để thông tin văn hóa đối với mạch trình hiện hành.
<b>CurrentInputLanguage</b>	Đi lấy hoặc đặt để ngôn ngữ nhập liệu hiện hành đối với mạch trình hiện hành.
<b>ProductName</b>	Đi lấy tên sản phẩm được gắn liền với ứng dụng này.
<b>ProductVersion</b>	Đi lấy phiên bản sản phẩm được gắn liền với ứng dụng này.
<b>StartupPath</b>	Đi lấy lối tìm về (path) đối với tập tin EXE cho khởi động ứng dụng.

Bạn để ý một vài thuộc tính chẳng hạn **CompanyName** và **ProductName** cho phép bạn tìm lại metadata cấp assembly. Ngoài ra, lớp **Application** còn định nghĩa một số tính huống như theo bảng 2-4:

**Bảng 2-4: Các tình huống của lớp Application**

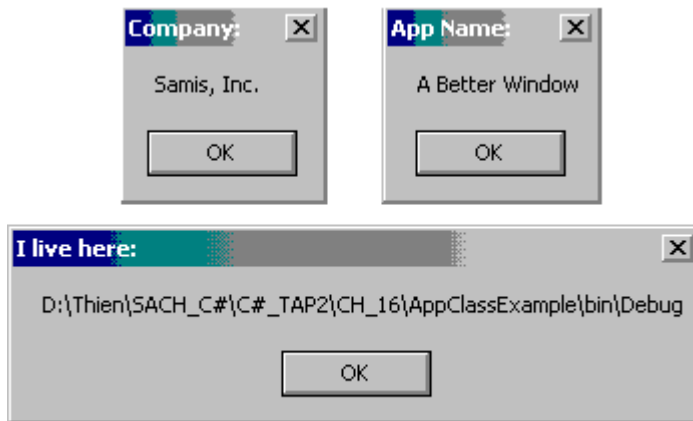
<b>ApplicationExit</b>	Tình huống xảy ra khi ứng dụng sắp sửa đóng lại.
<b>Idle</b>	Tình huống xảy ra khi ứng dụng kết thúc việc xử lý và sắp sửa bước vào giai đoạn “ngồi chơi xơi nước”.
<b>ThreadExit</b>	Tình huống xảy ra khi một mạch trình sắp đóng lại. Khi mạch trình chính đối với một ứng dụng đang sắp sửa đóng lại, tình huống này được gây ra trước tiên, theo sau bởi một tình huống ApplicationExit.

## 2.3.1 Thử thực hành với lớp Application

Để minh họa một vài chức năng của lớp Application (cũng như xem trước việc thụ lý tình huống Windows Forms) ta thử tăng cường MainWindow thô thiển bằng cách thực hiện các công việc sau đây:

- Cho hiển thị một vài thông tin cơ bản liên quan đến ứng dụng vào lúc khởi động.
- Phản ứng trước tình huống Application.Exit.
- Thực hiện một vài tiền xử lý của thông điệp WM\_LBUTTONDOWN.

Để bắt đầu, giả sử bạn nói rộng manifest bằng cách sử dụng một số attribute đánh dấu tên ứng dụng và công ty đã tạo ra nó. Bạn thêm hai hàng **[assembly:...]** vào đầu dự án sau các lệnh using. Bạn phải thêm **using System.Reflection;** để có thể sử dụng các attribute **[assembly:...]**. Hàm constructor của lớp được dẫn xuất từ Form có thể sử dụng thông tin này bằng cách sử dụng các thuộc tính của lớp **Application**, được hiển thị bằng cách sử dụng hàm **Show()** của lớp **MessageBox**. Sau đây là toàn bộ đoạn mã của dự án AppClassExemple, và hình 2-6 là kết xuất chạy chương trình:



Hình 2-06:Trích thông tin sử dụng lớp Application

```
using System;
using System.Windows.Forms;
using System.Reflection;
[assembly:AssemblyCompany("Samis, Inc.")]
[assembly:AssemblyProduct("A Better Window")]

namespace AppClassExample
{
    public class MainWindow: Form
    {
        public MainWindow()
    }
}
```

```

    {   GetStats();   // lấy thông tin thống kê
    }
    private void GetStats()
    {   MessageBox.Show(Application.CompanyName, "Company:");
        MessageBox.Show(Application.ProductName, "App Name:");
        MessageBox.Show(Application.StartupPath, "I live here:");
    }

    public static int Main(string[] args)
    {   Application.Run(new MainWindow());
        return 0;
    }
}

```

Phần in đậm là mới thêm vào. Khi bạn cho chạy ứng dụng này, bạn thấy những khung thông điệp khác nhau xuất hiện như theo hình 2-6:

## 2.3.2 Phản ứng trước tình huống ApplicationExit

Bây giờ ta thử cấu hình hoá biểu mẫu có khả năng phản ứng trước tình huống **ApplicationExit**. Nếu bạn muốn chặn hứng tình huống **ApplicationExit**, đơn giản bạn cho đăng ký một hàm hành sự custom với delegate sử dụng tác tử +=:

```

public class MainForm: Form
{
    public MainForm()
    {
        ...
        // chặn hứng tình huống ApplicationExit
        Application.ApplicationExit += new EventHandler(Form_OnExit);
    }
    // Hàm thụ lý tình huống
    private void Form_OnExit(object sender, EventArgs evArgs)
    {   MessageBox.Show("Không xong rồi!", "Ứng dụng nghèo rồi...");
    }
}

```

Bạn để ý dấu ấn của hàm thụ lý tình huống **ApplicationExit** phải tuân thủ một kiểu dữ liệu delegate **System.EventHandler**:

```

// Nhiều tình huống GUI sử dụng loại delegate này (EventHandler)
// đòi hỏi 2 thông số
public delegate void EventHandler(object sender, EventArgs e);

```

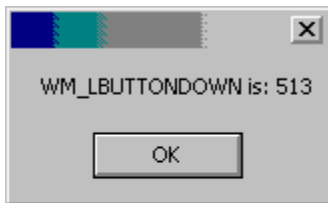
Thông số thứ nhất **sender** thuộc kiểu dữ liệu **System.Object**, tượng trưng cho đối tượng đã gọi đi tình huống. Còn thông số **e**, thuộc kiểu dữ liệu **EventArgs** (hoặc một hậu duệ của nó) chứa thông tin có ý nghĩa đối với tình huống hiện hành. Thí dụ, nếu bạn có



hàm thụ lý tình huống đáp ứng tình huống con chuột, thì thông số **MouseEventArgs** sẽ chứa những chi tiết liên quan đến con chuột như tọa độ (x, y) của vị trí con nháy. Nếu bạn cho chạy ứng dụng kể trên, thì một khung thông điệp sẽ hiện lên cho biết chương trình chấm dứt.

### 2.3.3 Xử lý trước các thông điệp với lớp Application

Bước cuối cùng trong thí dụ của chúng tôi là tiến hành một vài xử lý logic thông điệp WM\_LBUTTONDOWN. Có thể bạn đã biết, thông điệp Windows chuẩn này được gọi đi khi nút trái trên con chuột bị ấn xuống trong lòng vùng client area đối với một biểu mẫu nào đó (hoặc với bất cứ ô control GUI nào đó được trang bị đáp ứng tình huống này). Bây giờ, bạn nên để ý là bạn có thể tìm thấy một thể thức đơn giản hơn để chặn hứng những tình huống chuẩn về con chuột về sau cuối chương này. Bước này của dự án hiện hành chỉ đơn giản muốn minh họa làm thế nào thực hiện bất cứ logic tiền xử lý trước khi tình huống được trọn vẹn xử lý.



**Hình 2-07: Sàng lọc các thông điệp.**

Khi bạn muốn sàng lọc (filter) các thông điệp trên NET Framework, công việc đầu tiên của bạn là tạo một lớp mới biết thi công giao diện **IMessageFilter**. Nó rất đơn giản vì giao diện này chỉ định nghĩa một hàm hành sự, **PreFilterMessage()**. Trả về “true” để sàng lọc thông điệp và ngăn không cho nó được gọi đi, hoặc “false” cho phép thông điệp tiếp tục đi.

Trong phạm vi thi công của mình, bạn có thể xem xét vùng mục tin vào **Message.Msg** để trích ra trị số của thông điệp Windows (trong trường hợp của chúng tôi, WM\_LBUTTONDOWN mang trị số 513). Sau đây là toàn bộ dự án bổ sung phân lọc các thông điệp. Phần in đậm liên quan đến việc lọc thông điệp mới bổ sung. Hình 2-7 cho biết kết quả:

```
using System;
using System.Windows.Forms;
using System.Reflection;
using Microsoft.Win32; // dùng lọc thông điệp
[assembly:AssemblyCompany("Samis")]
[assembly:AssemblyProduct("A Better Window")]
namespace MyToto
{
    // Tạo một message filter
    public class MyMessageFilter: IMessageFilter
    { public bool PreFilterMessage(ref Message_m)
      { // Chặn hứng thông điệp WM_LBUTTONDOWN = 513
        if (m.Msg == 513)
        { MessageBox.Show("WM_LBUTTONDOWN is: " + m.Msg);
          return true;
        }
        return false; // Bỏ qua các thông điệp khác
      }
    }
}
```

```

    }
}

public class MainWindow: Form
{
    private MyMessageFilter msgFilter = new MyMessageFilter();
    public MainWindow()
    {
        GetStats();
        Application.ApplicationExit += new
            EventHandler(Form_OnExit);
        Application.AddMessageFilter(msgFilter);
    }

    private void GetStats()
    {
        MessageBox.Show(Application.CompanyName, "Company");
        MessageBox.Show(Application.ProductName, "App Name:");
        MessageBox.Show(Application.StartupPath, "I live here:");
    }

    private void Form_OnExit(object sender, EventArgs evArgs)
    {
        MessageBox.Show("Thôi rồi", "Chết nghèo rồi");
        // Gỡ bỏ cái lọc thông điệp
        Application.RemoveMessageFilter(msgFilter);
    }

    public static int Main(string[] args)
    {
        Application.Run(new MainWindow());
        return 0;
    }
}
}

```

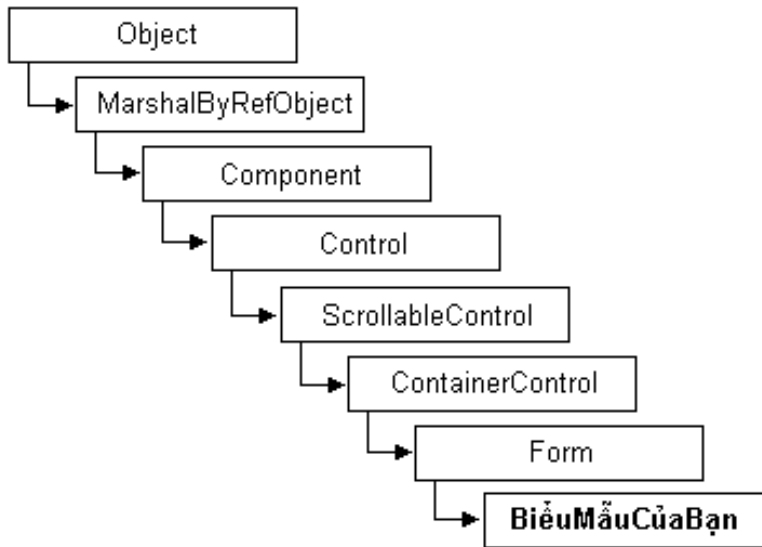
Hình 2-07 cho thấy kết quả khi bạn click nút trái con chuột trên mặt bằng biểu mẫu, bất cứ chỗ nào.

## 2.4 Hình thù một biểu mẫu ra sao?

Như vậy, bạn đã biết vai trò của đối tượng **Application**. Tiếp theo, bạn nên tìm hiểu chức năng của lớp **Form**. Thế biểu mẫu là gì?

Biểu mẫu là một màn hình, thường là hình chữ nhật, mà bạn có thể trình bày thông tin cho người sử dụng cũng như chấp nhận người sử dụng khổ vào dữ liệu. Biểu mẫu có thể là những cửa sổ chuẩn, cửa sổ MDI (multiple document interface), khung đối thoại, v.v.. Cách dễ nhất định nghĩa giao diện người sử dụng đối với một biểu mẫu là cho đặt lên mặt bằng biểu mẫu những ô control. Biểu mẫu theo lập trình thiên đối tượng là những đối tượng mang những thuộc tính (phác họa dáng dấp của biểu mẫu), những hàm hành sự (định nghĩa hành vi của biểu mẫu) và tính hướng (định nghĩa mối tương tác với người sử dụng). Bằng cách cho đặt để những thuộc tính và viết những đoạn mã đáp ứng tình

huống, bạn xem như uốn nắn (customize) “cắt ni” đối tượng đáp ứng đòi hỏi của ứng dụng.



**Hình 2-08: Sơ đồ dẫn xuất của kiểu dữ liệu Form**

được dùng tạo biểu mẫu. Lớp này được nhận một số lớn chức năng từ những lớp khác trên chuỗi kế thừa, như theo hình 2-8. Ngoài ra, khuôn giá (framework) cho phép bạn kế thừa từ những biểu mẫu hiện hữu để thêm chức năng hoặc thay đổi hành vi hiện hành. Khi bạn thêm một biểu mẫu vào dự án, bạn có thể chọn liệu xem nó kế thừa từ lớp **Form** cung cấp bởi framework hoặc từ biểu mẫu mà bạn đã tạo ra trước đó.

Muốn đi sâu vào chi tiết mô tả từng thành viên của mỗi lớp kể trên cũng phải mất một quyển sách dày cộm. Điều quan trọng là bạn chỉ nên tìm hiểu hành vi chủ chốt của từng lớp. Sau đó khi cần điều gì thì bạn lục xem chi tiết trên MSDN. Bây giờ, ta lần lượt tìm hiểu chức năng cơ bản của từng lớp trên hình 2-8.

## 2.4.1 Lớp Object, MarshalByRefObject

Bạn nên nhớ cho là giống như bất cứ lớp nào trong thế giới .NET, lớp **Form** được dẫn xuất từ **System.Object**. Lớp **MarshalByRefObject** định nghĩa hành vi cho kiểu dữ liệu **Form** này từ xa theo qui chiếu, thay vì theo trị. Do đó, nếu từ xa bạn cho thể hiện một đối tượng **Form**, thì xem ra bạn đang thao tác trên một qui chiếu về đối tượng **Form** trên một máy vi tính đặt nằm ở xa, chứ không phải trên một bản sao tại chỗ của **Form**.

Giống như với tất cả các đối tượng trong .NET Framework, các biểu mẫu được xem như là những thể hiện lớp **Form**. Biểu mẫu mà bạn tạo ra thông qua Windows Forms Designer là một lớp được dẫn xuất từ lớp **System.Windows.Forms.Form**, và khi bạn cho hiển thị một thể hiện của biểu mẫu vào lúc chạy, lớp này là khuôn mẫu

## 2.4.2 Lớp Component

Lớp cơ bản đầu tiên mà bạn quan tâm là **Component**. Kiểu dữ liệu **Component** cung cấp việc thi công cơ bản một giao diện **IComponent**. Giao diện này định nghĩa sẵn một thuộc tính **Site**, và thuộc tính này lại trả về một giao diện khác mang tên **ISite** (lạ nhỉ!). Ngoài ra, giao diện **IComponent** kế thừa một tính hướng duy nhất từ giao diện **IDisposable** mang tên là **Disposed**:

```
public interface IComponent: IDisposable
{
    // thuộc tính Site
    public ISite Site {virtual get; virtual set}
    // tính hướng Disposed
    public event EventHandler Disposed;
}
```

Còn giao diện **ISite** lại định nghĩa một số hàm hành sự cho phép một đối tượng **Control** tương tác với container “chủ chứa” (thí dụ một Form là chủ chứa của một ô control Button).

```
public interface ISite: IServiceProvider
{
    // các thuộc tính của giao diện ISite
    public IComponent Component { virtual get; }
    public IContainer Container { virtual get; }
    public bool DesignMode { virtual get; }
    public string Name { virtual get; virtual set; }
}
```

Nếu bạn muốn tạo một ô control “cây nhà lá vườn” cần được thao tác vào lúc thiết kế, (thường được gọi là custom control, mà chúng tôi sẽ đề cập đến trong tập III bộ sách này), thì có lẽ bạn sẽ quan tâm đến những thuộc tính được định nghĩa bởi giao diện **ISite**.

Ngoài thuộc tính **Site**, lớp **Component** còn cung cấp thi công hàm hành sự **Dispose()**. Chắc bạn còn nhớ, ta chỉ triệu gọi hàm **Dispose()** khi một component không còn được sử dụng nữa. Thí dụ, khi một **Form** sắp sửa bị đóng lại, thì hàm **Dispose()** sẽ tự động được gọi vào cho **Form** cũng như đối với tất cả các ô control khác nằm trên biểu mẫu. Bạn hoàn toàn tự do cho phủ quyết hàm này trong lớp dẫn xuất của bạn để giải phóng những nguồn lực vào thời gian thích ứng cũng như gỡ bỏ những qui chiếu về các đối tượng khác để có thể thu hồi ký ức thông qua dịch vụ garbage collector (GC).

```
public override void Dispose()
{
    base.Dispose();
    // bạn làm gì đó ở đây...
}
```

Thí dụ sau đây minh họa việc thi công các giao diện **ISite**, **IComponent** và **IContainer** dùng trong một thư viện chủ chứa (library container) đơn giản. Thí dụ này dùng đến các namespace: **System**, **System.ComponentModel**, và **System.Collections**

```
// Thi công giao diện ISite.

// Lớp ISBNSite tượng trưng cho tên ISBN của component sách
class ISBNSite: ISite
{
    private IComponent m_curComponent;
    private IContainer m_curContainer;
    private bool m_bDesignMode;
    private string m_ISBNCmpName;

    public ISBNSite(IContainer actvCntr, IComponent prntCmpnt)
    {
        m_curComponent = prntCmpnt;
        m_curContainer = actvCntr;
        m_bDesignMode = false;
        m_ISBNCmpName = null;
    }

    // Thi công giao diện ISite.
    public virtual IComponent Component
    {
        get {return m_curComponent; }
    }

    public virtual IContainer Container
    {
        get {return m_curContainer; }
    }

    public virtual bool DesignMode
    {
        get {return m_bDesignMode;}
    }

    public virtual string Name
    {
        get {return m_ISBNCmpName;}
        set {m_ISBNCmpName = value;}
    }

    // Thi công giao diện IServiceProvider.
    public virtual object GetService(Type serviceType)
    { // trong thí dụ này ta không sử dụng đối tượng dịch vụ nào
        return null;
    }
}

// Lớp BookComponent tượng trưng cho book component của library
// container. Lớp này thi công giao diện IComponent.

class BookComponent: IComponent
{
    public event EventHandler Disposed;
    private ISite m_curISBNSite;
    private string m_bookTitle;
```

```
private string m_bookAuthor;

public BookComponent(string Title, string Author)
{
    m_curISBNSite = null;
    Disposed = null;
    m_bookTitle = Title;
    m_bookAuthor = Author;
}

public string Title
{
    get {return m_bookTitle;}
}

public string Author
{
    get {return m_bookAuthor;}
}

public virtual void Dispose()
{
    // không có gì phải dọn dẹp.
    if(Disposed != null)
        Disposed(this,EventArgs.Empty);
}

public virtual ISite Site
{
    get {return m_curISBNSite;}
    set {m_curISBNSite = value;}
}

public override bool Equals(object cmp)
{
    BookComponent cmpObj = (BookComponent)cmp;
    if(this.Title.Equals(cmpObj.Title) &&
        this.Author.Equals(cmpObj.Author))
        turn true;
    return false;
}

public override int GetHashCode()
{
    return base.GetHashCode();
}
}
```

## 2.4.3 Lớp Control

Lớp **Control** sẽ thiết lập những hành vi thông dụng mà bất cứ kiểu dữ liệu GUI nào đều đòi hỏi phải có. Các thành viên chủ chốt của lớp này cho phép bạn cấu hình kích thước và vị trí của ô control, trích ra mục quản HWND (window handle) cũng như thu nhận dữ liệu nhập từ bàn phím hoặc từ con chuột. Bảng 2-5 liệt kê một vài thuộc tính của lớp **Control** mà bạn phải quan tâm.

**Bảng 2-5: Các thuộc tính chủ chốt của lớp Control**

Các thuộc tính	Ý nghĩa
<b>Top, Left, Bottom, Right, Bounds, ClientRectangle, Height, Width</b>	Các thuộc tính này cho biết những đặc tính khác nhau liên quan đến kích thước hiện hành của đối tượng được dẫn xuất từ lớp <b>Control</b> . <b>Bounds</b> trả về một Rectangle (hình chữ nhật) cho biết kích thước của ô control. <b>ClientRectangle</b> trả về một Rectangle tương ứng với kích thước của vùng khách hàng (client area) của ô control.
<b>Created, Disposed, Enabled, Focused, Visible</b>	Các thuộc tính này trả về một trị bool cho biết tình trạng của ô control hiện hành.
<b>Handle</b>	Thuộc tính này trả về một mục quản (handle) HWND, một trị số tượng trưng cho mã nhận diện của ô control.
<b>ModifierKeys</b>	Thuộc tính static này cho kiểm tra lại tình trạng hiện hành của các phím Shift, Ctrl, Alt và trả về tình trạng theo kiểu dữ liệu <b>Key</b> .
<b>MouseButtons</b>	Thuộc tính static này cho kiểm tra lại tình trạng hiện hành của các nút chuột (nút left, right, middle) và trả về tình trạng theo kiểu dữ liệu <b>MouseButtons</b> .
<b>Parent</b>	Thuộc tính này trả về một đối tượng <b>Control</b> tượng trưng cho thân phận cha-mẹ của ô control hiện hành.
<b>TabIndex, TabStop</b>	Các thuộc tính này dùng đặt để thứ tự nhảy của phím Tab trên các ô control.
<b>Text</b>	Đây là dòng văn bản hiện hành được gắn liền với ô control này.

Ngoài ra, lớp **Control** còn định nghĩa một số hàm hành sự cho phép bạn tương tác với bất cứ ô control nào. Sau đây là một danh sách không đầy đủ các hàm hành sự của lớp **Control**, bảng 2-6:

**Bảng 2-6: Các hàm hành sự chủ chốt của lớp Control**

Các hàm hành sự	Ý nghĩa
<b>GetStyle(), SetStyle()</b>	Các hàm hành sự này dùng thao tác lên cờ hiệu style flag của ô control hiện hành sử dụng enumeration <b>ControlStyles</b> .

<b>Hide(), Show()</b>	Hai hàm hành sự này gián tiếp đặt để tình trạng của thuộc tính <b>Visible</b> .
<b>Invalidate()</b>	Hàm này ép ô control tự vẽ lại bằng cách ép một thông điệp vẽ vào hàng nối đuôi thông điệp. Hàm này bị phủ quyết cho phép bạn khai báo cụ thể một Rectangle phải cho tô điểm lại mặt mũi (refresh) thay vì toàn bộ vùng client area.
<b>OnXXXX()</b>	Lớp <b>Control</b> định nghĩa vô số hàm hành sự mà bạn có thể cho nạp chồng (overridden) bởi một subclass đáp ứng những tình huống khác nhau (chẳng hạn <b>OnMouseMove()</b> , <b>OnKeyDown()</b> , v.v..). Về sau, trong chương này, khi bạn muốn chặn hững một tình huống dựa trên GUI, bạn sẽ thấy có hai cách tiếp cận: cách thứ nhất là phủ quyết một trong những hàm thụ lý tình huống (event handler) hiện hữu. Cách thứ hai là thêm một custom event handler cho một delegate nào đó.
<b>Refresh()</b>	Hàm này ép ô control triệu gọi hàm <b>Invalidate()</b> và lập tức cho tô điểm lại ô control và bất cứ ô control con-cái nào.
<b>SetBound(), SetLocation(), SetClientArea()</b>	Các hàm hành sự này được dùng để thiết lập kích thước của các ô control.

### 2.4.3.1 Cho đặt để style của một biểu mẫu

Ta thử xem hai hàm hành sự khá lý thú của lớp **Control**: **GetStyle()** và **SetStyle()**. Bạn có khả năng thay đổi những style mặc nhiên của đối tượng **Form** nếu bạn thấy là cần thiết. Trước tiên, bạn kiểm tra lại enumeration **ControlStyles**:

```
public enum ControlStyles
{
    AllPaintingInWmPaint,
    CacheText,
    ContainerControl,
    EnableNotifyMessage,
    FixedHeight,
    FixedWidth,
    Opaque,
    ResizeRedraw,
    Selectable,
    StandardClick,
    StandardDoubleClick,
    SupportsTransparentBackColor,
```



```

        UserMouse,
        UserPaint
    }

```

Một đối tượng **Form** thường có một bộ style mặc nhiên được đặt để. Nếu bạn muốn khai báo nhiều style, bạn có thể sử dụng tác tử OR đối với các trị của enum **ControlStyle**. Giả sử, bạn có một **Form**, **StyleForm**, chứa một nút ấn duy nhất kiểu **Button**, **btnGetStyles**. Trên hàm thụ lý tình huống **Click** đối với nút ấn này, bạn có thể kiểm tra xem đối tượng **Form** này có hỗ trợ một style nào đó hay không bằng cách sử dụng hàm hành sự **GetStyle()** như sau:

```

// Cho thấy false
private void btnGetStyles_Click(object sender, System.EventArgs e)
{
    MessageBox.Show(GetStyle(ControlStyles.ResizeRedraw).ToString(),
        "bạn có ResizeRedraw hay không ?");
}

```

Muốn đặt để một style về false/true đối với một ô control nào đó, bạn có thể viết:

```

public StyleForm()
{
    . . .
    SetStyle(ControlStyles.ResizeRedraw, true);
}

```

**ResizeRedraw** là một trị mà bạn muốn hiển hình thêm vào một biểu mẫu nào đó. Theo mặc nhiên style này không hiện dịch (active) và do đó, một đối tượng **Form** sẽ không tự động tự tô vẽ lại khi nó bị thay đổi kích thước. Đây có nghĩa là nếu bạn chặn hứng một tình huống **Paint**, và thay đổi kích thước **Form**, việc tô vẽ của bạn không được “làm tươi” (refresh) lại. Nếu bạn muốn bảo đảm tình huống **Paint** sẽ được phát ra bất cứ lúc nào người sử dụng thay đổi kích thước của **Form**, phải bảo đảm là bạn khai báo style **ResizeRedraw** sử dụng **SetStyle()**. Một cách khác là chặn hứng tình huống **Resize** của lớp **Form** và triệu gọi trực tiếp hàm **Invalidate()**:

```

private void StyleForm_Resize(object sender, System.EventArgs e)
{
    Invalidate(); // hàm này ép tô điểm lại..
}

```

Hiện hình là có thể bạn muốn chặn hứng tình huống **Resize()** khi bạn có những công việc cần làm thêm ngoài việc kích hoạt một châu tô điểm.

Để minh họa tác dụng của việc đặt để **Form style**, giả sử bạn có một đoạn mã **GDI+** lo vẽ một đường lăm chấm bao quanh hình chữ nhật vùng client area, như sau:

```

this.Paint += new
    System.Windows.Forms.PaintEventHandler(this.StyleForm_Paint);

private void StyleForm_Paint(object sender,
    System.Windows.Forms.PaintEventArgs e)

```

```

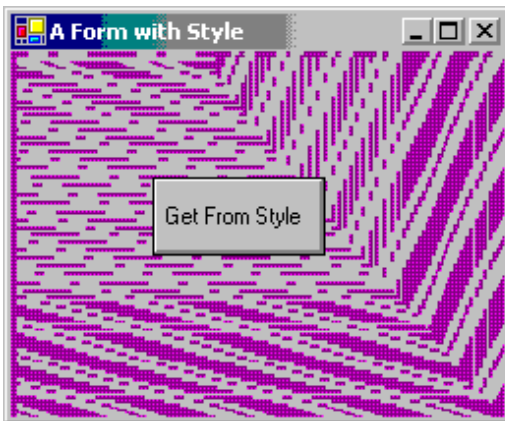
{
    Graphics g = e.Graphics;
    Pen customDashPen = new Pen(Color.BlueViolet, 5);
    float[] myDashes = {5.0f, 2.0f, 1.0f, 3.0f};
    customDashPen.DashPattern = myDashes;
    g.DrawRectangle(customDashPen, ClientRectangle);
}

```

Và nếu `ResizeRedraw = false`, thì bạn sẽ thấy hình bên trái hình 2-09, còn nếu bạn cho lệnh

```
SetStyle(ControlStyles.ResizeRedraw, true);
```

và cho chạy lại chương trình, thì bạn có hình đúng đắn hơn là hình tay phải trên hình 2-09.



**Hình 2-09: `ResizeRedraw = false`**



**`ResizeRedraw = true`**

Sau đây là một thí dụ khác với dự án mang tên **SimpleFormApp** được kết sinh bởi Visual Studio .NET IDE. Bạn thử xem nó làm gì.

```

namespace SimpleFormApp
{
    using System;
    using System.Drawing;
    using System.Collections;
    using System.ComponentModel;
    using System.Windows.Forms;
    using System.Data;

    public class MainForm: System.Windows.Forms.Form
    {
        private System.ComponentModel.Container components;

        public MainForm()    //hàm contructor
        {
            InitializeComponent();
            BackColor = Color.LemonChiffon;    // Màu nền Background.

```

```

        Text = "My Fantastic Form";           // Tựa đề của Form.
        Size = new Size(200, 200);           // kích thước 200 * 200.
        CenterToScreen();                     // Canh biểu mẫu nằm giữa màn hình
    }

    protected override void Dispose(bool disposing)
    {
        if(disposing)
        {
            if (components != null)
            {
                components.Dispose();
            }
        }
        base.Dispose(disposing);
    }

    private void InitializeComponent()
    {
        this.AutoScaleBaseSize = new System.Drawing.Size(5, 13);
        this.ClientSize = new System.Drawing.Size(292, 273);
        this.Text = "Form1";
        this.Resize += new System.EventHandler(
                                this.MainForm_Resize);
        this.Paint += new System.Windows.Forms.PaintEventHandler(
                                this.MainForm_Paint);
    }

    [STAThread]
    static void Main()
    {
        Application.Run(new MainForm());
    }

    private void MainForm_Resize(object sender, System.EventArgs e)
    {
        Invalidate();
    }

    private void MainForm_Paint(object sender,
        System.Windows.Forms.PaintEventArgs e)
    {
        Graphics g = e.Graphics;
        g.DrawString("Windows Forms is for building GUIs!",
            new Font("Times New Roman", 20),
            new SolidBrush(Color.Black),
            this.DisplayRectangle); // hiển thị trên client rect.
    }
}

```

## 2.4.4 Điều khiển các tình huống

Lớp **Control** cũng định nghĩa một số tình huống mà ta có thể chia thành hai nhóm: Mouse events, tình huống liên quan đến con chuột và Keyboard events, tình huống liên quan đến bàn phím. Bảng 2-7 cho liệt kê các tình huống của lớp **Control**:

**Bảng 2-7: Các tình huống chủ yếu của lớp Control**

Các tình huống	Ý nghĩa
<b>Click, DoubleClick, MouseEnter, MouseLeave, MouseDown, MouseUp, MouseMove, MouseHover, MouseWheel</b>	Các tình huống này liên quan đến con chuột.
<b>KeyPress, KeyUp, KeyDown</b>	Các tình huống này liên quan đến bàn phím.

Bây giờ ta thử làm việc với lớp **Control**. Để minh họa việc sử dụng vài thành viên cốt lõi, trước tiên bạn tạo một Form mới, cũng cho gọi là MainForm, cung cấp những chức năng sau đây:

- Cho kích thước ban đầu của biểu mẫu về những trị hồ đồ nào đó
- Cho phủ quyết hàm **Dispose()**
- Phản ứng trước những tình huống **MouseMove** và **MouseUp** (sử dụng hai cách tiếp cận)
- Chặn hứng và xử lý dữ liệu nhập từ bàn phím

Để bắt đầu, giả sử bạn có một lớp C# mới được dẫn xuất từ **Form**. Trước tiên, cho nhập tu hàm constructor mặc nhiên để chỉnh tọa độ top, left, bottom và right của biểu mẫu sử dụng các thuộc tính khác nhau của lớp **Control**. Để xác minh những thay đổi này, bạn sử dụng thuộc tính **Bound**, rồi cho hiển thị kích thước hiện hành theo dạng chuỗi. Bạn để ý thuộc tính **Bound** trả về một kiểu **Rectangle** được định nghĩa trong namespace **System.Drawing**. Do đó, bảo đảm đặt đề một qui chiếu assembly về **System.Drawing.dll**, nếu bạn dựng biểu mẫu này bằng tay (Visual Studio .NET sẽ tự động thêm vào đối với những dự án Windows Forms).

```
// Cần phải có đối với định nghĩa Rectangle
using System.Drawing;
...
public class MainForm: Form
{
    public MainForm()
    {
        Top = 100;
        Left = 75;
        Height = 100;
        Width = 500;
        MessageBox.Show(Bound.ToString(), "Rectangle hiện hành");
    }
    public static int Main(string [] args)
```

```

    { Application.Run(new MainForm());
      return 0;
    }
}

```

Khi bạn cho chương trình chạy, hình 10-A hiện lên xác định tọa độ của biểu mẫu của bạn. Một khi bạn ấn OK cho biến cửa sổ thông điệp, thì cửa sổ chính dài ngoẵng hiện lên (hình 10-B).



(A)



(B)



(C)

Hình 2-10: (A) Thuộc tính Bounds (B) Các thuộc tính Top, Left, Height và Width (C) Chặn hứng tình huống MouseUp

Bước tiếp theo là cho phủ quyết hàm hành sự được kế thừa **Component.Dispose()**. Như bạn còn nhớ, đối tượng **Application** có một tình huống mang tên **Application.Exit**. Nếu bạn cấu hình biểu mẫu của bạn cho chặn hứng tình huống này, thì coi như bạn thực thụ báo tin cho hủy ứng dụng. Như là một phương án thay thế đơn giản hơn, bạn có thể đi đến kết quả như nhau bằng cách đơn giản phủ quyết hàm hành sự abstract **Dispose()**. Tuy nhiên, trước đó bạn phải triệu gọi hàm **Dispose()** của lớp cơ bản:

```

public class MainForm: Form
{
    ...
    // Các dự án V.S .NET Windows Forms tự động
    // hỗ trợ hàm hành sự này
}

```

```

    public override void Dispose()
    {
        base.Dispose();
        MessageBox.Show("Dispose biểu mẫu này...")
    }
}

```

### 2.4.4.1 Đáp ứng tình huống **MouseUp** - Phần 1

Bước tiếp theo là cho chặn hứng tình huống **MouseUp**. Mục tiêu là cho hiển thị tọa độ (x, y) khi mà tình huống **MouseUp** xảy ra. Khi bạn muốn đáp ứng trước tình huống ngay trên ứng dụng Windows Form, nhìn chung bạn có hai cách tiếp cận. Cách thứ nhất quá quen thuộc đối với bạn là dùng delegate. Cách tiếp cận thứ hai là cho phủ quyết một hàm hành sự lớp cơ bản thích ứng. Bây giờ, ta thử xem hai cách tiếp cận vừa kể trên, bắt đầu sử dụng delegate. Sau đây là MainForm được nhập tu .

```

// Cần phải có đối với định nghĩa Rectangle
using System.Drawing;
...
public class MainForm: Form
{
    public MainForm()
    {
        Top = 100;
        Left = 75;
        Height = 100;
        Width = 500;
        MessageBox.Show(Bound.ToString(), "Rectangle hiển hành");

        // Nghe ngóng tình huống MouseUp .....
        this.MouseUp += new MouseEventHandler (OnMouseUp);
    }

    // hàm hành sự được triệu gọi đáp ứng tình huống MouseUp
    public void OnMouseUp(object sender, MouseEventArgs e)
    {
        this.Text = "Click tại: (" + e.X + ", " + e.Y + ")";
    }

    public static int Main(string [] args)
    {
        Application.Run(new MainForm());
        return 0;
    }
}

```

Bạn cho chạy thử chương trình sau khi thêm đoạn mã kể trên. Hình 10-C cho thấy kết quả khi bạn di chuyển con chuột.

Bạn còn nhớ, delegate thường nhận **EventArgs** như là thông số thứ hai. Khi bạn thụ lý tình huống di chuyển chuột, thì thông số thứ hai thuộc kiểu **MouseEventArgs**. Kiểu dữ liệu này, được định nghĩa trong namespace **System.Windows.Forms**, định nghĩa một số thuộc tính khá lý thú mà ta có thể dùng làm thông kê liên quan đến tình trạng con chuột, như theo bảng 2-8 sau đây:

**Bảng 2-8: Các thuộc tính lớp MouseEventArgs**

<b>Button</b>	Cho biết nút nào trên con chuột bị ấn xuống. Trị của <b>Button</b> được chỉ định bởi enumeration <b>MouseButtons</b> (gồm None, Middle, Left, Right, XButton1, và XButton2).
<b>Clicks</b>	Cho biết số lần nút chuột bị ấn xuống và nhả ra.
<b>Delta</b>	Cho biết số đếm có dấu của số lần detent “bánh xe” con chuột quay. Một detent là một “bánh cóc” con chuột.
<b>X</b>	Cho biết tọa độ x của một mouse click.
<b>Y</b>	Cho biết tọa độ y của một mouse click.

Phần thi công hàm hành sự **OnMouseUp()** đơn giản chỉ trích tọa độ (x,y) của vị trí con nháy chuột rồi cho hiển thị tọa độ này lên tựa đề của biểu mẫu thông qua thuộc tính **Text** được kế thừa. Xem hình 2-10-C.

Nếu muốn, bạn cũng có thể chặn hứng một tình huống **MouseMove** và cho hiển thị tọa độ của vị trí dữ liệu lên tựa đề của biểu mẫu. Như vậy, vị trí hiện hành của con nháy chuột có thể được theo dõi trong khuôn viên của client area.

```
public class MainForm: Form
{
    public MainForm()
    {
        Top = 100;
        Left = 75;
        Height = 100;
        Width = 500;
        MessageBox.Show(Bound.ToString(), "Rectangle hiện hành");

        // Nghe ngóng các tình huống MouseUp, MouseMove .....
        this.MouseUp += new EventHandler(OnMouseUp);
        this.MouseMove += new EventHandler(OnMouseMove);
    }

    // hàm hành sự được triệu gọi đáp ứng tình huống MouseUp
    public void OnMouseUp(object sender, MouseEventArgs e)
    {
        MessageBox.Show("Xì tốp! Đừng click em nữa!");
    }

    // hàm hành sự được triệu gọi đáp ứng tình huống MouseMove
    public void OnMouseMove(object sender, MouseEventArgs e)
    {
        this.Text = "Vị trí hiện tại: (" + e.X + ", " + e.Y + ")";
    }
}
```

```

    }
    public static int Main(string [] args)
    {
        Application.Run(new MainForm());
        return 0;
    }
    ...
}

```

#### 2.4.4.2 Xác định xem nút nào trên con chuột bị click

Bạn nên nhớ cho là tình huống **MouseUp** (hoặc **MouseDown**) được phát ra bất cứ lúc nào một nút nào đó trên con chuột bị bấm tắt. Muốn biết chính xác nút nào (phải, trái hoặc giữa) bạn cần xét đến thuộc tính **Button** của lớp **MouseEventArgs**. Xem bảng 2-8. Trị của **Button** được chỉ định bởi enumeration **MouseButtons**.

```

// hàm hành sự được triệu gọi đáp ứng tình huống MouseUp
public void OnMouseUp(object sender, MouseEventArgs e)
{
    // Nút nào bị click?
    if(e.Button == MouseButtons.Left)
        MessageBox.Show("Left Click!");
    else if(e.Button == MouseButtons.Right)
        MessageBox.Show("Right Click!");
    else
        MessageBox.Show("Middle Click!");
}

```

Bạn thử cho chạy lại, rồi click lên nút trái, nút phải, v.v.. rồi xem messagebox hiện lên thế nào.

#### 2.4.4.3 Đáp ứng các tình huống Mouse Events - Phần 2

Cách tiếp cận thứ hai là chặn hứng những tình huống trên một kiểu dữ liệu được dẫn xuất từ lớp **Control** rồi cho phủ quyết hàm hành sự lớp cơ bản thích ứng, trong trường hợp này là **OnMouseUp()** hoặc **OnMouseMove()**. Lớp **Control** có định nghĩa một số hàm hành sự protected và virtual; các hàm sẽ tự động được triệu gọi khi tình huống tương ứng được kích hoạt (triggered). Nếu bạn sử dụng kỹ thuật này, bạn sẽ không cần thao tác đặc biệt thông qua một hàm thụ lý tình huống “cây nhà lá vườn”. Thí dụ:

```

public class MainForm: Form
{
    public MainForm()
    {
        ...
        // không cần các lệnh này khi override, cho comment out
        // this.MouseUp += new MouseEventHandler(OnMouseUp);
        // this.MouseMove += new MouseEventHandler(OnMouseMove);
    }
}

```



```
// hàm hành sự được triệu gọi đáp ứng tình huống MouseUp
protected override void OnMouseUp(
    /*object sender,*/ MouseEventArgs e)
{
    // Nút nào bị click?
    if(e.Button == MouseButtons.Left)
        MessageBox.Show("Left Click!");
    else if(e.Button == MouseButtons.Right)
        MessageBox.Show("Right Click!");
    else
        MessageBox.Show("Middle Click!");
}

// hàm hành sự được triệu gọi đáp ứng tình huống MouseMove
protected override void OnMouseMove(
    /*object sender,*/ MouseEventArgs e)
{
    this.Text = "Vị trí hiện tại: (" + e.X + ", " + e.Y + ")";
}

public static int Main(string [] args)
{
    Application.Run(new MainForm());
    return 0;
}
...
}
```

Bạn để ý là dấu ấn của các hàm hành sự này chỉ nhận một thông số kiểu **MouseEventArgs**, thay vì hai thông số như theo delegate **MouseEventHanler**. Nếu bạn cho chạy lại chương trình, thì không có gì thay đổi. Thế là tốt rồi!. Điển hình là bạn cần phủ quyết một hàm hành sự "OnXXXX()" chỉ khi nào bạn có những công việc phải làm thêm trước khi tình huống được phát đi. Cách tiếp cận được ưa thích nhất (và là cách duy nhất mà Visual Studio .NET sử dụng đến) là thụ lý trực tiếp tình huống như bạn đã làm trong thí dụ đầu tiên về con chuột.

## 2.4.5 Đáp ứng các tình huống Keyboard

Xử lý việc nhập liệu thông qua bàn phím hầu như cũng tương tự như với hoạt động của con chuột. Đoạn mã sau đây chặn hứng tình huống **KeyUp** và cho hiển thị lên message box tên của ký tự được gõ vào. Bạn sử dụng kỹ thuật delegate. Có một hàm hành sự mang tên **OnKeyUp()** mà bạn có thể phủ quyết như là phương án thay thế delegate.

```
public class MainForm: Form
{
    public MainForm()
    {
        Top = 100;
        Left = 75;
```

```

        Height = 100;
        Width = 500;
        MessageBox.Show(Bound.ToString(), "Rectangle hiện hành");

        // Nghe ngóng các tình huống MouseUp, MouseMove .....
        this.MouseUp += new MouseEventHandler(OnMouseUp);
        this.MouseMove += new MouseEventHandler(OnMouseMove);

        // Lắng theo tình huống KeyUp
        this.KeyUp += new KeyEventHandler(OnKeyUp);
    }

    // hàm hành sự được triệu gọi đáp ứng tình huống MouseUp
    public void OnMouseUp(object sender, MouseEventArgs e)
    {
        MessageBox.Show("Xì tốp! Đừng click em nữa!");
    }

    // hàm hành sự được triệu gọi đáp ứng tình huống MouseMove
    public void OnMouseMove(object sender, MouseEventArgs e)
    {
        this.Text = "Vị trí hiện tại: (" + e.X + ", " + e.Y + ")";
    }

    // hàm hành sự được triệu gọi đáp ứng tình huống KeyUp
    public void OnKeyUp(object sender, KeyEventArgs e)
    {
        MessageBox.Show(e.KeyCode.ToString(), "Phím bị ấn xuống!");
    }

    public static int Main(string [] args)
    {
        Application.Run(new MainForm());
        return 0;
    }
    ...
}

```

Như bạn có thể thấy, kiểu dữ liệu **KeyEventArgs** duy trì một enumeration **KeyCode** cho biết mã nhận diện ID của phím bị ấn xuống. Ngoài ra, lớp **KeyEventArgs** cũng định nghĩa một số thuộc tính rất hữu ích, như theo bảng 2-9 sau đây:

**Bảng 2-9: Các thuộc tính của lớp KeyEventArgs**

<b>Alt</b>	Đi lấy một trị cho biết liệu xem phím ALT có bị ấn xuống hay không.
<b>Control</b>	Đi lấy một trị cho biết liệu xem phím CTRL có bị ấn xuống hay không.
<b>Handled</b>	Đi lấy/Đặt để một trị cho biết liệu xem tình huống được thụ lý hay chưa.
<b>KeyCode</b>	Đi lấy mã phím đối với một tình huống <b>KeyDown</b> hoặc <b>KeyUp</b> .
<b>KeyData</b>	Đi lấy dữ liệu phím (key data) đối với một tình huống <b>KeyDown</b> hoặc <b>KeyUp</b> .
<b>KeyValue</b>	Đi lấy trị phím (keyboard value) đối với một tình huống <b>KeyDown</b> hoặc <b>KeyUp</b> .

- Modifiers** Đi lấy modifier flags đối với một tình huống **KeyDown** hoặc **KeyUp**. Việc này cho biết tổ hợp phím modifier keys (CTRL, SHIFT, và ALT) nào được ấn xuống.
- Shift** Đi lấy một trị cho biết liệu xem phím SHIFT có bị ấn xuống hay không.

## 2.4.6 Lớp Control - Phần chót

Lớp **Control** còn định nghĩa thêm một số thuộc tính cho phép bạn cấu hình màu sắc, phông chữ, chức năng lõi-thả và trình đơn (menu). Ngoài ra, nó còn cung cấp những hành vi liên quan đến việc neo đậu (anchoring) và cập bến (docking) cửa sổ thể nào. Nhưng có lẽ nhiệm vụ quan trọng nhất của lớp **Control** là cơ chế tô điểm hình ảnh, văn bản và đồ họa trên mặt bằng client area thông qua hàm hành sự **OnPaint()**. Trước tiên, bạn xem qua danh sách bổ sung các thuộc tính của lớp **Control**, bảng 2-10:

**Bảng 2-10: Danh sách bổ sung các thuộc tính của lớp Control**

Thuộc tính	Ý nghĩa
<b>AllowDrop</b>	Nếu AllowDrop được cho về true, thì ô control này cho phép có những tác vụ lõi thả và dùng đến các tình huống.
<b>Anchor</b>	Thuộc tính này ấn định góc cạnh nào trên ô control sẽ được neo vào góc cạnh của container.
<b>BackColor, BackgroundImage, Font, ForeColor, Cursor</b>	Các thuộc tính này dùng cấu hình mặt bằng client area sẽ được hiển thị thế nào.
<b>ContextMenu</b>	Thuộc tính này cho biết trình đơn shortcut (còn gọi là context menu, hoặc pop-up menu) sẽ hiện lên khi người sử dụng right-click lên ô control.
<b>Dock</b>	Thuộc tính này kiểm soát việc ô control này sẽ neo ở góc cạnh nào của ô chứa (container). Thí dụ, khi được phép neo ở trên đỉnh container, thì ô control sẽ được hiển thị nằm trên đỉnh container và bành ra theo chiều dài của container.
<b>Opacity</b>	Thuộc tính này xác định độ mờ của ô control, tính theo % (0.0 là hoàn toàn trong suốt, 1.0 là hoàn toàn mờ cảm).
<b>Region</b>	Thuộc tính này cấu hình một đối tượng <b>Region</b> , cho biết outline/silhouette/boundary của ô control.

<b>RightToLeft</b>	Thuộc tính này được dùng trong những ứng dụng mang tính quốc tế theo đây ngôn ngữ được viết từ phải sang trái.
--------------------	--

Ngoài ra, lớp **Control** còn định nghĩa một số hàm hành sự và tình huống dùng tạo cấu hình làm thế nào ô control đáp ứng trước những tác vụ lỗi thả cũng như tô điểm. Bảng 2-11 cho thấy những hàm hành sự và tình huống của lớp **Control** bổ sung.

Bây giờ ta thử minh họa việc sử dụng các thành viên bổ sung của lớp **Control**. Ta cho màu nền về màu cà chua, độ trong suốt 50%, và con nháy chuột ở chế độ chờ đợi (hình đồng hồ cát). Điều quan trọng nhất là giải quyết tình huống **Paint** cho hiện lên dòng văn bản “Ta in ra gì đây?” trên vùng client area của biểu mẫu.

**Bảng 2-11: Danh sách bổ sung các hàm hành sự của lớp Control**

Hàm hành sự / Tình huống	Mô tả
<b>OnDragDrop()</b> <b>OnDragEnter()</b> <b>OnDragLeave()</b> <b>OnDragOver()</b>	Các hàm hành sự này được dùng điều khiển việc lôi-thả đối với một <b>Control</b> hậu duệ nào đó.
<b>ResetFont()</b> <b>ResetCursor()</b> <b>ResetForeColor()</b> <b>ResetBackColor()</b>	Các hàm hành sự này cho reset những attribute UI khác nhau của một ô control con-cái về trị tương ứng của ô control cha-mẹ.
<b>OnPaint()</b>	Các lớp kế thừa phải phủ quyết hàm hành sự này để thụ lý tình huống <b>Paint</b> .
<b>DragEnter event</b> <b>DragLeave event</b> <b>DragDrop event</b> <b>DragOver event</b>	Các tình huống này được phát đi đáp ứng các tác vụ lôi-thả.
<b>Paint event</b>	Tình huống này sẽ được phát đi bất cứ lúc nào ô Control trở thành “phôi phai” cần phải được tô điểm lại.

```
using System;
using System.Drawing; // cần thiết đối với Color, Brush và Font
using System.Collections;
using System.ComponentModel;
using System.Windows.Forms;
using System.Data;

namespace WindowsApplication1
{
```

```

public class MainForm: Form
{
    ...

    public MainForm()
    {
        InitializeComponent();

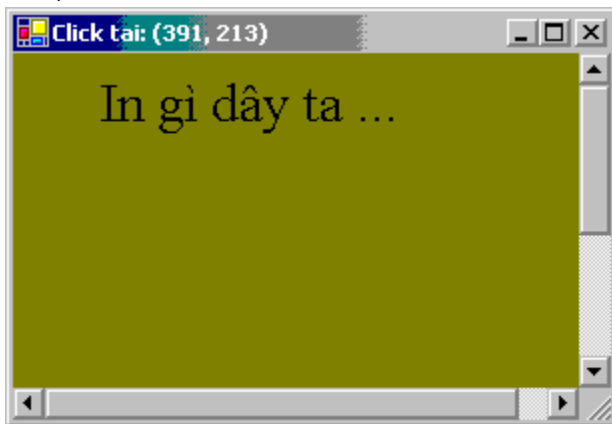
        // Cho đặt để vài thuộc tính được kế thừa từ Lớp Control
        BackColor = Color.Tomato; // cho về màu cà chua
        Opacity = 0.50d;           // độ trong suốt 50%
        this.Cursor = Cursors.WaitCursor; // hình đồng hồ cát

        // thụ lý tình huống Paint
        this.Paint += new PaintEventHandler(MainForm_Paint);
    }

    private void MainForm_Paint(object sender, PaintEventArgs e)
    {
        Graphics g = e.Graphics;
        g.DrawString("In gì đây ta ...",
                     new Font("Times New Roman", 20),
                     new SolidBrush(Color.Black), 40, 10);
    }
    ...
    private void InitializeComponent()
    {
        this.components = new System.ComponentModel.Container();
        this.Size = new System.Drawing.Size(300,300);
        this.Text = "Form1";
    }

    public static int Main(string[] args)
    {
        Application.Run(new MainForm());
        return 0;
    }
}

```



Hình 2-11: Vẽ với thuộc tính Opacity

Nếu bạn cho chạy chương trình trên, bạn sẽ thấy một biểu mẫu mang màu cà chua trong suốt với dòng chữ: “In gì đây ta ...”. Xem hình 2-11.

### 2.4.6.1 Căn bản về tô vẽ (painting)

Khía cạnh quan trọng nhất trong thí dụ trên là việc thụ lý tình huống **Paint**. Bạn nhớ cho việc sử dụng delegate định nghĩa một hàm hành sự chấp nhận một thông số kiểu **PaintEventArgs**. Kiểu này định nghĩa hai thuộc tính giúp bạn tạo cấu hình cho châu tô điểm hiện hành đối với ô Control như theo bảng 2-12:

**Bảng 2-12: Các thuộc tính của lớp PaintEventArgs**

<b>ClipRectangle</b>	Cho biết hình chữ nhật mà ta sẽ tô điểm vào đây. Thuộc tính này là read-only.
<b>Graphics</b>	Cho biết đối tượng <b>Graphics</b> dùng để tô điểm. Thuộc tính này là read-only.

Thuộc tính quan trọng của lớp **PaintEventArgs** là **Graphics**, được triệu gọi để tìm thấy một đối tượng **Graphics** dùng trong châu tô điểm. Bạn sẽ làm quen đối tượng này khi chúng tôi đề cập đến GDI+ tại chương 3, “Tìm hiểu đồ họa và GDI+”, tập III bộ sách này. Tạm thời bạn biết cho là lớp **Graphics** định nghĩa một số thành viên cho phép bạn hiển thị văn bản, các hình vẽ hình học, hình ảnh lên một ô Control.

Cuối cùng trong thí dụ trên bạn đã sử dụng đến thuộc tính **Cursor** để hiển thị một đồng hồ cát bất cứ lúc nào con nháy chuột còn nằm trong khuôn viên của client area. Kiểu **Cursors** có thể được gán cho bất cứ thành viên của enumeration **Cursors** (nghĩa là **Arrow**, **Cross**, **UpCross**, **Help**, v.v.):

```
public MainForm()
{
    . . .
    this.Cursor = Cursors.WaitCursor;
}
```

## 2.4.7 Lớp ScrollableControl

Lớp **ScrollableControl** là một lớp cơ bản định nghĩa một số hàm thành viên cho phép những ô control hỗ trợ những thanh điều hành (scrollbar) ngang hoặc dọc. Thuộc tính khó hiểu nhất là **AutoScroll** và thuộc tính liên đới **AutoScrollMinSize**. Thí dụ, giả sử bạn muốn bảo đảm rằng nếu người sử dụng thay đổi lại kích thước biểu mẫu, các thanh điều hành ngang dọc sẽ tự động được thêm vào nếu kích thước client area nhỏ thua hoặc bằng 300\*300 pixels. Về mặt lập trình, nhiệm vụ của bạn khá đơn giản:

```
// Hai dòng lệnh sau đây có thể đưa vào hàm constructor
// của lớp hoặc vào hàm InitializeComponent()
// bạn để ý phải qui chiếu namespace System.Drawing để
// truy xuất kiểu dữ liệu Size.
this.AutoScroll = true;
this.AutoScrollMinSize = new System.Drawing.Size (300,300);
```

Lớp **ScrollableControl** sẽ lo mọi việc. Thí dụ, nếu bạn có một biểu mẫu chứa một số đối tượng con-cái (button, label, v.v..) bạn sẽ thấy logic điều hành bảo đảm trọn mặt bằng biểu mẫu có thể nhìn thấy được. Với mấy dòng lệnh kể trên đưa vào hàm constructor, rồi cho chạy lại bạn sẽ thấy kết quả là hình 2-11 ở trên.

Lớp **ScrollableControl** định nghĩa một số thành viên ngoài hai thuộc tính vừa kể trên, nhưng không nhiều lắm. Ngoài ra, nếu bạn muốn hoàn toàn điều khiển tiến trình điều hành, bạn có khả năng tạo và thao tác những kiểu ScrollBar riêng rẽ (chẳng hạn HScrollBar và VScrollBar). Bạn nên tham khảo MSDN tìm hiểu những thành viên còn lại của lớp **ScrollableControl**.

## 2.4.8 Lớp ContainerControl

Lớp **ContainerControl** định nghĩa việc hỗ trợ quản lý focus của một ô control GUI nào đó. Trong thực tế, hành vi được định nghĩa bởi **System.Windows.Forms.ContainerControl** sẽ hữu ích hơn khi bạn đang xây dựng một biểu mẫu chứa một số ô control con-cái với mong muốn dành cho người sử dụng quyền sử dụng đến phím Tab để điều khiển qua lại focus. Sử dụng đến một số nhỏ thành viên lớp **ContainerControl**, bạn có thể, về mặt lập trình, nhận được ô control hiện được tuyển chọn, hoặc ép một ô control khác nhận focus, v.v.. Bảng 2-13 liệt kê một số thành viên của lớp **ContainerControl**.

**Bảng 2-13: Liệt kê các thành viên của lớp ContainerControl**

Thành viên lớp	Ý nghĩa
<b>ActiveControl, ParentForm</b>	Các thuộc tính này cho phép bạn nhận lấy và cho đặt để ô control hiện dịch (active control), cũng như tìm lại qui chiếu chỉ về biểu mẫu chứa chấp (host) ô control này.
<b>ProcessTabKey()</b>	Hàm này cho phép bạn theo lập trình hiện dịch phím <Tab> để chuyển focus qua ô control kế tiếp có sẵn.

Bạn nhớ cho tất cả các “hậu duệ” của **System.Windows.Forms.Control** đều kế thừa các thuộc tính **TabStop** và **TabIndex**. Như bạn có thể đoán ra các thuộc tính này cho đặt để một thứ tự “viếng thăm” của phím <Tab> được duy trì bởi một container cha-mẹ, và được dùng phối hợp với những thành viên được cung cấp bởi lớp **ContainerControl**. Chúng ta sẽ trở lại vấn đề trật tự Tab khi nói đến lập trình các ô control, trên chương 6, tập III bộ sách .NET toàn tập này, “Windows Forms Controls”.

## 2.4.9 Lớp Form

Lớp **Form** được xem như là lớp cơ bản trực tiếp để tạo ra những lớp Form “cây nhà lá vườn” của bạn. Ngoài vô số thành viên được kế thừa từ các lớp **Control**, **ScrollableControl** và **ContainerControl**, lớp **Form** còn bổ sung một số khá lớn chức năng. Ta thử bắt đầu với những thuộc tính cốt lõi. Xem bảng 2-14.

**Bảng 2-14: Các thuộc tính cốt lõi của lớp Form**

Các thuộc tính	Ý nghĩa
<b>AcceptButton</b>	Đi lấy hoặc đặt để button lên biểu mẫu; button này sẽ bị click khi người sử dụng ấn phím <Enter>.
<b>ActiveMDIChild</b> <b>IsMDIChild</b> <b>IsMDIContainer</b>	Các thuộc tính này được dùng trong lòng phạm trù của một ứng dụng MDI (Multiple Document Interface).
<b>AutoScale</b>	Đi lấy hoặc đặt để một trị cho biết liệu xem biểu mẫu sẽ điều chỉnh kích thước của nó thích ứng với phong chữ dùng đến trên biểu mẫu và điều chỉnh theo tỉ lệ (scale) những ô control của biểu mẫu.
<b>BorderStyle</b>	Đi lấy hoặc đặt để kiểu đường viền (border style) của biểu mẫu. Thuộc tính này được dùng phối hợp với enumeration <b>FormBorderStyle</b> .
<b>CancelButton</b>	Đi lấy hoặc đặt để nút control cần phải bị click khi người sử dụng ấn phím <Esc>.
<b>ControlBox</b>	Đi lấy hoặc đặt để một trị cho biết liệu xem biểu mẫu có một control box hay không.
<b>Menu, MergedMenu</b>	Đi lấy hoặc đặt để trình đơn (được trộn lại) đối với biểu mẫu.
<b>MaximizeBox</b> <b>MinimizeBox</b>	Các thuộc tính này được dùng để xác định liệu xem biểu mẫu này có trang bị những ô phình to hoặc ô teo lại.
<b>ShowInTaskBar</b>	Biểu mẫu này có nên cho hiện lên trên thanh task bar Windows hay không.
<b>StartPosition</b>	Đi lấy hoặc đặt để vị trí xuất phát của biểu mẫu lúc chạy, như được kê khai bởi enumeration <b>FormStartPosition</b> .



<b>WindowState</b>	Thuộc tính này sẽ cấu hình các biểu mẫu sẽ được hiển thị vào lúc khởi động (startup). Thuộc tính được dùng phối hợp với enumeration <b>FormWindowState</b> .
--------------------	--

Lớp **Form** không bổ sung nhiều về mặt hàm hành sự, vì phần lớn các chức năng của biểu mẫu được kế thừa từ những lớp cơ bản mà chúng ta đã biết qua. Tuy nhiên, bảng 2-15 cũng liệt kê một số hàm hành sự được bổ sung của lớp **Form** mà bạn nên quan tâm.

**Bảng 2-15: Các hàm hành sự bổ sung của lớp Form**

Các hàm hành sự	Ý nghĩa
<b>Activate()</b>	Cho hiện dịch một đối tượng Form nào đó.
<b>Close()</b>	Cho đóng lại biểu mẫu.
<b>CenterToScreen()</b>	Cho biểu mẫu nằm giữa màn hình.
<b>LayoutMDI</b>	Sắp xếp mỗi biểu mẫu con-cái (theo khai báo bởi enumeration <b>LayoutMDI</b> ) trong lòng biểu mẫu cha-mẹ.
<b>OnResize()</b>	Hàm này có thể bị phủ quyết đáp ứng các tình huống <b>Resize</b> .
<b>ShowDialog()</b>	Cho hiển thị một biểu mẫu modal.

Cuối cùng lớp **Form** cũng định nghĩa một số tình huống mà bạn nên quan tâm đến. Bảng 2-16 liệt kê các tình huống của lớp **Form**.

**Bảng 2-16: Các tình huống được tuyển chọn của lớp Form**

Các tình huống	Ý nghĩa
<b>Activated</b>	Xảy ra khi biểu mẫu được hiện dịch bởi chương trình hoặc bởi người sử dụng, nghĩa là khi biểu mẫu được chuyển lên đầu màn hình trên một ứng dụng hiện dịch.
<b>Closed</b> <b>Closing</b>	Các tình huống này được dùng để xác định xem khi nào một biểu mẫu sắp sửa bị đóng lại hoặc đã bị đóng.
<b>MDIChildActivate</b>	Xảy ra khi một biểu mẫu MDI con-cái được hiện dịch trong lòng một ứng dụng MDI.

### 2.4.9.1 Thử chơi một tí với lớp Form



Hình 2-12:: Sử dụng lớp Form

Tới đây coi như bạn đã biết sơ sơ những chức năng mà lớp **Form** cung cấp. Bây giờ ta thử tạo một biểu mẫu sử dụng đến những thành viên khác nhau trên chuỗi kế thừa.

Thí dụ sau đây là một đối tượng **Form** hiện lên nằm chính giữa màn hình. Ngoài ra, tính hướng điều chỉnh kích thước **Resize** được thụ lý. Chỉ cần triệu gọi hàm **Invalidate()** để tô điểm lại (refresh) client area. Theo cách này, chuỗi chữ “Windows Form dùng xây giao diện GUI!” sẽ nằm khít trong lòng hình chữ nhật biểu mẫu (bạn để ý việc sử dụng đến thuộc tính **DisplayRectangle**).

```
namespace SimpleFormApp
{
    using System;
    using System.Drawing;
    using System.Collections;
    using System.ComponentModel;
    using System.Windows.Forms;
    using System.Data;

    public class MainForm: System.Windows.Forms.Form
    {
        private System.ComponentModel.Container components;

        public MainForm()
        {
            InitializeComponent();
            BackColor = Color.LemonChiffon;    // Background color
            Text = "My Fantastic Form";        // Tựa đề Form
            Size = new Size(200, 200);        // 200 * 200
            CenterToScreen();                 // Canh nằm giữa màn hình
        }

        protected override void Dispose(bool disposing)
        {
            if(disposing)
            {
                if (components != null)
                {
                    components.Dispose();
                }
            }
            base.Dispose(disposing);
        }

        #region Windows Form Designer generated code
        private void InitializeComponent()
        {
            this.AutoScaleBaseSize = new System.Drawing.Size(5, 13);
            this.ClientSize = new System.Drawing.Size(292, 273);
            this.Text = "Form1";
        }
    }
}
```

```

        this.Resize += new System.EventHandler(this.MainForm_Resize);
        this.Paint += new System.Windows.Forms.PaintEventHandler(
            this.MainForm_Paint);
    }
#endregion

static void Main()
{
    Application.Run(new MainForm());
}

private void MainForm_Resize(object sender, System.EventArgs e)
{
    Invalidate();
}

private void MainForm_Paint(object sender,
                             System.Windows.Forms.PaintEventArgs e)
{
    Graphics g = e.Graphics;
    g.DrawString("Windows Forms dùng xây giao diện GUI!",
        new Font("Times New Roman", 20),
        new SolidBrush(Color.Black),
        this.DisplayRectangle); // Cho hiển thị client rectangle
}
}

```

Hình 2-12 (trang trước) cho thấy kết quả chạy chương trình trên.:

## 2.5 Tạo những trình đơn với Windows Forms

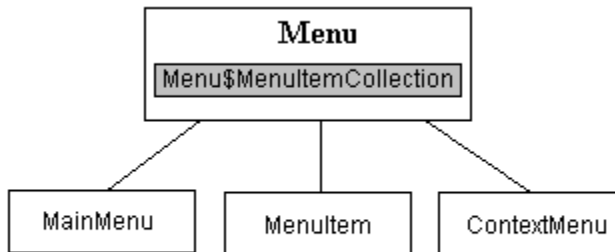
Bước kế tiếp của ta là học cách thiết lập một hệ thống trình đơn cho phép người sử dụng tương tác với ứng dụng. Namespace **System.Windows.Forms** cung cấp vô số lớp cho phép bạn tạo trình đơn chính nằm trên đầu biểu mẫu và những trình đơn shortcut. Để bắt đầu, ta thử viết một trình đơn chính cho phép người sử dụng thoát khỏi ứng dụng sử dụng đến lệnh thông dụng **File | Exit**. Xem hình 2-13.

Lớp đầu tiên bạn phải quan tâm là **System.Windows.Forms.Menu**, hoạt động như là lớp cơ bản đối với những lớp khác liên hệ đến trình đơn, và là một lớp abstract, nghĩa là bạn không thể hiện lộ trực tiếp một thể hiện của lớp này. Thay vào đó, bạn phải tạo những thể hiện của một hoặc nhiều lớp được dẫn xuất từ lớp cơ bản này. Lớp **Menu** định nghĩa những hành vi tập trung vào trình đơn, chẳng hạn cho phép truy xuất vào từng mục trình đơn (menu item), trộn trình đơn (đối với ứng dụng MDI), đóng trình đơn lại, v.v.. Hình 2-14 cho thấy mối liên hệ giữa các lớp cốt lõi:



Hình 2-13: Một hệ thống trình đơn đơn giản

Lớp **Menu** định nghĩa một lớp nằm lòng mang tên **Menu\$MenuItemCollection**, mà các lớp **MainMenu**, **MenuItem** và **ContextMenu** sẽ kế thừa. Collection này sẽ chứa một lô những mục chọn trình đơn có liên hệ được truy xuất sử dụng đến thuộc tính **Menu.Items**. Lớp **Menu** định nghĩa những thành viên cốt lõi như sau, như theo bảng liệt kê 2-17:



Hình 2-14: Cây đẳng cấp của Windows.Forms.Menu

Bảng 2-17: Các thành viên của lớp Menu

Các thành viên	Ý nghĩa
<b>Handle</b>	Thuộc tính này cho phép truy xuất mục quản (handle) HMENU tượng trưng cho trình đơn này.
<b>IsParent</b>	Thuộc tính này cho biết liệu xem trình đơn này có chứa bất cứ mục chọn trình đơn nào hay không hoặc là mục chọn cao nhất.
<b>MdiListItem</b>	Thuộc tính này trả về MenuItem chứa danh sách các cửa sổ MDI con-cái.
<b>MenuItems</b>	Thuộc tính này trả về một thể hiện của lớp

	<b>Menu.MenuItem.Collection</b> nằm lồng, tượng trưng cho trình đơn con sở hữu bởi lớp dẫn xuất từ lớp <b>Menu</b> .
<b>GetMainMenu()</b>	Hàm này trả về mục chọn <b>MainMenu</b> chứa trình đơn này.
<b>MergeMenu()</b>	Cho trộn lại với nhau các mục chọn trình đơn khác với trình đơn được khai báo bởi các thuộc tính <b>mergeType</b> và <b>mergeOrder</b> . Hàm này được dùng để trộn một trình đơn trên MDI container với trình đơn của một MDI con-cái hiện dịch.
<b>CloneMenu()</b>	Hàm này cho đặt để trình đơn này giống y chang một trình đơn khác.

## 2.5.1 Lớp Menu\$MenuItemCollection

Có thể nói thành viên quan trọng nhất của lớp **Menu** là thuộc tính **MenuItems**, trả về một thể hiện của lớp **Menu\$MenuItemCollection** nằm lồng. Bạn nhớ lại các lớp nằm lồng có thể hữu ích khi bạn muốn thiết lập một mối liên hệ logic giữa các lớp liên đới. Ở đây, lớp **Menu\$MenuItemCollection** tượng trưng cho tập hợp của tất cả các submenu thuộc quyền cai quản của một đối tượng **Menu** được dẫn xuất.

Thí dụ, nếu bạn tạo một **MainMenu** để tượng trưng cho trình đơn “File” cao nhất, có thể bạn muốn thêm những đối tượng **MenuItems** (thí dụ Open, Save, Close, Save As) vào collection. Như bạn có thể chờ đợi, lớp **Menu\$MenuItemCollection** định nghĩa những thành viên lo việc thêm vào hoặc gỡ bỏ những đối tượng kiểu **MenuItem**, hoặc đếm xem số **MenuItem** hiện hành cũng như truy xuất vào một mục tin trong collection. Bảng 2-18 liệt kê một vài thành viên cốt lõi của lớp **Menu\$MenuItemCollection** nằm lồng.

**Bảng 2-18: Các thành viên của lớp Menu\$MenuItemCollection**

Các thành viên	Ý nghĩa
<b>Count</b>	Thuộc tính này cho biết số lượng <b>MenuItems</b> nằm trong collection.
<b>Add()</b> <b>AddRange()</b> <b>Remove()</b>	Các hàm này thêm vào (hoặc gỡ bỏ) một <b>MenuItem</b> mới vào collection. Bạn để ý hàm hành sự <b>Add()</b> bị nạp chồng (overloaded) nhiều lần cho phép bạn khai báo shortcut key, delegate, và gì gì nữa đó. <b>AddRange()</b> chỉ hữu ích khi hàm cho phép bạn thêm một bản dãy <b>MenuItem</b> chỉ với một lệnh.
<b>Clear()</b>	Hàm này xoá sạch các <b>MenuItem</b> khỏi collection.

<b>Contains()</b>	Hàm này dùng xác định liệu xem một <b>MenuItem</b> nào đó có trong collection hay không .
-------------------	---

## 2.6 Tạo một hệ thống trình đơn

Sau khi hiểu xong các chức năng của lớp abstract **Menu** và lớp **Menu\$MenuItemCollection**, bạn có thể bắt đầu tạo một trình đơn **File** đơn giản bằng tay. Ta bắt đầu tạo một đối tượng **MainMenu**. Lớp **MainMenu** tượng trưng cho tập hợp những mục chọn trình đơn cao nhất (chẳng hạn File, Edit, View, Tools, Help, v.v.). Như vậy:

```
public class MainForm: Form
{
    // Main menu của Form
    private MainMenu mainMenu;

    public MainForm()           // hàm constructor
    { // Tạo main menu
        mainMenu = new MainMenu();
    }
    . . .
}
```

Một khi bạn đã có đối tượng **MainMenu**, bạn có thể dùng hàm hành sự **Menu\$MenuItemCollection.Add()** để thêm mục chọn trình đơn cao nhất (nghĩa là “File”). Hàm này trả về một lớp **MenuItem** mới tượng trưng cho trình đơn **File** mới được chèn vào.

Muốn chèn một subitem (chẳng hạn Exit), bạn chèn phụ thêm MenuItems vào **Menu\$MenuItemCollection** được duy trì bởi File MenuItem. Cuối cùng, khi bạn kết thúc việc tạo hệ thống trình đơn của mình, bạn cho gắn nó vào Form “giám quản” bằng cách dùng thuộc tính **Menu**:

```
public class MainForm: Form
{
    // Main menu của Form
    private MainMenu mainMenu;

    public MainForm()           // hàm constructor
    { // Tạo main menu
        mainMenu = new MainMenu();

        // Tạo File menu và thêm vào collection
        MenuItem miFile = mainMenu.MenuItems.Add("&File");

        // Bây giờ thêm một submenu Exit và thêm vào File menu.
```

```
// Phiên bản Add() này lấy:
// 1. một MenuItem mới
// 2. một delegate mới (EventHandler)
// 3. một phím shortcut tùy chọn
miFile.MenuItems.Add(new MenuItem(
    "E&xit",
    new EventHandler(this.FileExit_Clicked),
    Shortcut.CtrlX));

// Gắn main menu vào đối tượng Form
this.Menu = mainMenu;
}
. . .
}
```

Bạn để ý nếu bạn chen một “&” trong lòng tên mục chọn, thí dụ “&File” chẳng hạn, thì mẫu tự đi sau sẽ được gạch dưới, và người sử dụng sẽ khó tổ hợp phím <Alt+mẫu tự có gạch dưới> để truy xuất mục chọn này, thí dụ <ALT+F>, thì trình đơn File sẽ được khởi động. Trong tổ hợp phím <Alt+F>, <F> được gọi là access key.

Khi bạn thêm mục chọn submenu Exit, bạn khai báo một shortcut tùy chọn, **Shortcut.CtrlX** chẳng hạn. Enumeration **System.Windows.Forms.Shortcut** có đầy đủ chi tiết trên MSDN, cho biết những phím shortcut thông thường (Ctrl+C, Ctrl+V, F1, F2, Ins, v.v..) và vô số tổ hợp khác.

Bạn để ý là bạn có thể cho đặt để thuộc tính **BackColor** của biểu mẫu bằng cách dùng thành viên **MainMenu.GetForm()**. **GetForm()** cho biết biểu mẫu nào chứa chấp trình đơn MainMenu. Sau đây là thí dụ đơn giản về trình đơn:

```
// Một ứng dụng trình đơn đơn giản
public class MainForm: Form
{
    private MainMenu mainMenu; // trình đơn chính của biểu mẫu

    public MainForm() // hàm constructor
    {
        // Cho đặt để các thuộc tính ban đầu của Form
        Text = "Simple Menu";
        CenterToScreen();

        // tạo main menu
        mainMenu = new MainMenu();

        // tạo trình đơn 'File | Exit'
        MenuItem miFile = mainMenu.MenuItems.Add("&File");
        miFile.MenuItems.Add(new MenuItem("E&xit",
            new EventHandler(this.FileExit_Clicked),
            Shortcut.CtrlX));

        // Gắn main menu vào đối tượng Form
        this.Menu = mainMenu;
    }
}
```

```

        // MainMenu.GetForm() trả về qui chiếu về owning Form
        mainMenu.GetForm().BackColor = Color.Black;
    }

    // File | Exit handlers.
    private void FileExit_Clicked(object sender, EventArgs e)
    {
        this.Close();    // đóng lại ứng dụng
    }

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

```

Hình 2-13 (trang 157) cho thấy kết quả việc thêm vào trình đơn File với mục chọn Exit.

## 2.6.1 Thêm một mục chọn trình đơn chớp bu khác

Bây giờ bạn muốn thêm một mục chọn trình đơn “Help” chỉ chứa duy nhất một subitem mang tên “About”. Ta cứ dựa theo logic của mô hình **File | Exit**: bắt đầu thêm một MenuItem mới vào đối tượng MainMenu (“Help”). Từ đây, lại thêm một subitem mới (“About”). Thí dụ đoạn mã thêm vào như sau:

```

// Tạo một trình đơn 'Help | About'.
MenuItem miHelp = mainMenu.MenuItems.Add("Help");
miHelp.MenuItems.Add(new MenuItem("&About",
                                new EventHandler(this.HelpAbout_Clicked),
                                Shortcut.CtrlA));
. . .

// Hàm thụ lý tình huống Help | About Menu item
private void HelpAbout_Clicked(object sender, EventArgs e)
{
    MessageBox.Show("Thí dụ về trình đơn ...");
}

```

Hình 2-13 (trang 157) cho thấy kết quả, thêm vào một trình đơn Help và About.



## 2.6.2 Tạo một trình đơn shortcut

Bây giờ ta quay qua xét đến việc tạo một trình đơn gọi là shortcut menu (hoặc còn gọi là pop-up menu) khi người sử dụng right-click lên một mục tin nào đó. Lớp **ContextMenu** tượng trưng cho loại trình đơn này. Giống như trên tiến trình tạo MainMenu, việc của bạn là thêm riêng rẽ từng **MenuItem** vào **MenuItemCollection** để tượng trưng cho những subitem có khả năng được tuyển chọn. Lớp sau đây sử dụng đến pop-up menu để cho phép người sử dụng cấu hình kích thước phông chữ của một chuỗi chữ hiện lên trên client area:

```
namespace PopUpMenu
{
    using System;
    using System.Drawing;
    using System.Collections;
    using System.ComponentModel;
    using System.Windows.Forms;
    using System.Data;

    // Cấu trúc cho biết kích thước phông chữ: to, nhỏ, thường
    internal struct TheFontSize
    {
        public static int Huge = 30;
        public static int Normal = 20;
        public static int Tiny = 8;
    }

    public class MainForm: Form
    {
        // kích thước hiện hành phông chữ
        private int currFontSize = TheFontSize.Normal;

        // trình đơn popup của biểu mẫu
        private ContextMenu popUpMenu;

        // Dùng theo dõi current checked item.
        private MenuItem currentCheckedItem;
        private MenuItem checkedHuge;
        private MenuItem checkedNormal;
        private MenuItem checkedTiny;

        public MainForm()
        {
            // Cấu hình biểu mẫu lúc ban đầu
            Text = "PopUp Menu";
            CenterToScreen();

            // Trước tiên tạo một context menu
            popUpMenu = new ContextMenu();

            // Thêm các subitems rồi gán context menu
            // vào đối tượng biểu mẫu
            popUpMenu.MenuItems.Add("Huge", new
```

```

        EventHandler(PopUp_Clicked));
    popUpMenu.MenuItems.Add("Normal", new
        EventHandler(PopUp_Clicked));
    popUpMenu.MenuItems.Add("Tiny", new
        EventHandler(PopUp_Clicked));

    this.ContextMenu = popUpMenu;

    checkedHuge = this.ContextMenu.MenuItems[0];
    checkedNormal = this.ContextMenu.MenuItems[1];
    checkedTiny = this.ContextMenu.MenuItems[2];
    currentCheckedItem = checkedNormal;
    currentCheckedItem.Checked = true;

    // thụ lý tình huống Resize và Paint
    this.Resize += new
        System.EventHandler(this.MainForm_Resize);
    this.Paint += new PaintEventHandler(this.MainForm_Paint);
}

// PopUp_Clicked | X menu item handler
private void PopUp_Clicked(object sender, EventArgs e)
{
    currentCheckedItem.Checked = false;
    // Tìm ra tên chuỗi của mục tin được chọn
    MenuItem miClicked = (MenuItem)sender;
    string item = miClicked.Text;

    if(item == "Huge")
    {
        currFontSize = TheFontSize.Huge;
        currentCheckedItem = checkedHuge;
    }

    if(item == "Normal")
    {
        currFontSize = TheFontSize.Normal;
        currentCheckedItem = checkedNormal;
    }

    if(item == "Tiny")
    {
        currFontSize = TheFontSize.Tiny;
        currentCheckedItem = checkedTiny;
    }
    currentCheckedItem.Checked = true;
    Invalidate();
}

// hàm thụ lý tình huống Paint
private void MainForm_Paint(object sender, PaintEventArgs e)
{
    Graphics g = e.Graphics;

    // Vẽ văn bản gì đó trên client rect.
    g.DrawString("Click em xem nào ...",
        new Font("Times New Roman", (float)currFontSize),
        new SolidBrush(Color.Black),
        this.DisplayRectangle);
}

```

```

    }

    // hàm thụ lý tình huống Resize
    private void MainForm_Resize(object sender, System.EventArgs e)
    {
        Invalidate();
    }

    // Hàm Main()
    static void Main()
    {
        Application.Run(new MainForm());
    }
}

```

Bạn để ý là khi bạn thêm những subitems vào **ContextMenu**, bạn gán cho mỗi mục cùng hàm thụ lý tình huống, ở đây là **Popup\_Clicked()**. Khi một mục chọn bị click, thì **Popup\_Clicked()** được gọi vào. Sử dụng đến đối tượng “sender” bạn có thể xác định tên của **MenuItem** (nghĩa là chuỗi văn bản đã được gán trước đó) và tiến hành hành động thích ứng.

Ngoài ra, một khi bạn đã tạo một **ContextMenu**, bạn gán nó cho biểu mẫu thông qua thuộc tính **Control.ContextMenu**. Thí dụ, bạn có thể tạo một đối tượng **Button** trên một khung đối thoại đáp ứng đối với một trình đơn popup đặc biệt nào đó. Theo cách này, trình đơn shortcut chỉ hiển thị nếu đối tượng **Button** bị click trong khuôn viên hình chữ nhật của nút.

## 2.6.3 Tô điểm thêm hệ thống trình đơn của bạn

Lớp **MenuItem** cũng định nghĩa một số thành viên cho phép bạn cho hiệu lực, hoặc cất giấu một mục chọn trình đơn nào đó. Bảng 2-19 liệt kê một số thuộc tính của lớp **MenuItem** mà bạn nên quan tâm:

**Bảng 2-19: Các thuộc tính của lớp MenuItem**

Các thuộc tính	Ý nghĩa
<b>Checked</b>	Đi lấy hoặc đặt để một trị cho biết liệu xem một dấu check có xuất hiện cạnh bên văn bản của mục chọn trình đơn hay không.
<b>DefaultItem</b>	Đi lấy hoặc đặt để một trị cho biết liệu xem mục chọn trình đơn là mặc nhiên hay không.
<b>Enabled</b>	Đi lấy hoặc đặt để một trị cho biết liệu xem mục chọn trình đơn có hiệu lực hay không.
<b>Index</b>	Đi lấy hoặc đặt để một trị cho biết vị trí của mục chọn trình đơn trên trình đơn cha-mẹ của nó.

<b>MergeOrder</b>	Đi lấy hoặc đặt để một trị cho biết vị trí tương đối của mục chọn trình đơn khi nó được trộn chung với một mục chọn khác.
<b>MergeType</b>	Đi lấy hoặc đặt để một trị cho biết hành vi của mục chọn trình đơn này khi trình đơn của nó được trộn lại với một mục chọn khác.
<b>OwnerDraw</b>	Đi lấy hoặc đặt để một trị cho biết liệu xem đoạn mã mà bạn cung cấp sẽ vẽ mục chọn trình đơn hay là Windows sẽ vẽ mục chọn trình đơn.
<b>RadioCheck</b>	Đi lấy hoặc đặt để một trị cho biết liệu xem MenuItem, nếu bị checked, sẽ hiển thị một radio-button thay vì một check mark.
<b>Shortcut</b>	Đi lấy hoặc đặt để một trị cho biết shortcut key được gắn liền với mục chọn trình đơn.
<b>ShowShortcut</b>	Đi lấy hoặc đặt để một trị cho biết liệu xem shortcut key được gắn liền với mục chọn trình đơn sẽ được hiển thị cạnh tựa đề mục chọn trình đơn hay không.
<b>Text</b>	Đi lấy hoặc đặt để một trị cho biết tựa đề của mục chọn trình đơn.

Để minh họa vấn đề, ta thử nói rộng trình đơn popup kể trên cho hiển thị một check mark cạnh mục chọn trình đơn được tuyển. Bạn chỉ cần cho thuộc tính **Checked** về true là xong việc. Tuy nhiên, theo dõi mục chọn trình đơn nào phải được đánh dấu đòi hỏi bổ sung logic. Một cách tiếp cận khả thi là định nghĩa các đối tượng **MenuItem** một cách rõ ràng để theo dõi mỗi subitem và một MenuItem bổ sung tượng trưng cho mục chọn hiện được tuyển:

```
public class MainForm: Form
{
    // kích thước hiện hành phông chữ
    private int currFontSize = TheFontSize.Normal;

    // trình đơn popup của biểu mẫu
    private ContextMenu popUpMenu;

    // Dùng theo dõi current checked item.
    private MenuItem currentCheckedItem;
    private MenuItem checkedHuge;
    private MenuItem checkedNormal;
    private MenuItem checkedTiny;

    ...
}
```

Bước kế tiếp là gắn liền mỗi một những **MenuItem** này vào đúng subitem. Do đó, bạn phải nhậ tu hàm constructor như sau:

```
public MainForm()
{
    // Cấu hình biểu mẫu lúc ban đầu
    Text = "PopUp Menu";
```

```

CenterToScreen();

// Trước tiên tạo một context menu
popUpMenu = new ContextMenu();

// Thêm các subitems rồi gán context menu vào đối tượng biểu mẫu
popUpMenu.MenuItems.Add("Huge", new EventHandler(PopUp_Clicked));
popUpMenu.MenuItems.Add("Normal", new
    EventHandler(PopUp_Clicked));
popUpMenu.MenuItems.Add("Tiny", new EventHandler(PopUp_Clicked));

this.ContextMenu = popUpMenu;

// Cho mỗi MenuItem về đúng submenu
checkedHuge = this.ContextMenu.MenuItems[0];
checkedNormal = this.ContextMenu.MenuItems[1];
checkedTiny = this.ContextMenu.MenuItems[2];

// Bây giờ cho đánh dấu mục chọn trình đơn "Normal"
currentCheckedItem = checkedNormal;
currentCheckedItem.Checked = true;

```

Tới đây, bạn đã có cách nhận diện bằng chương trình mỗi subitem, cũng như mục chọn trình đơn được tuyển (khởi đi là ở Normal). Bước cuối cùng là nhậ tu hàm thụ lý tình huống **PopUp\_Clicked**, để có thể đánh dấu đúng MenuItem như theo chọn lựa của người sử dụng. Xem hình 2-15, kết quả thử nghiệm:

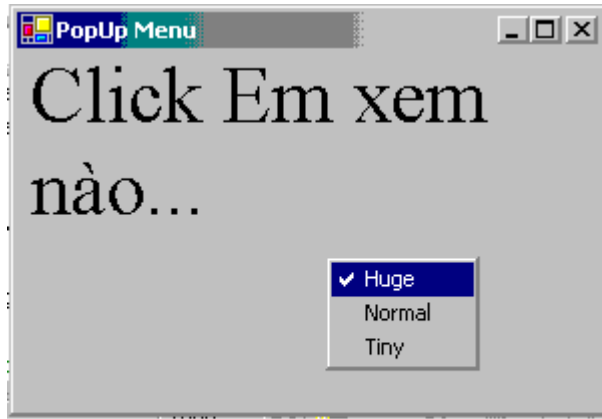
```

// PopUp_Clicked | X menu item handler
private void PopUp_Clicked(object sender, EventArgs e)
{
    // Uncheck mục chọn hiện được tuyển
    currentCheckedItem.Checked = false;

    // Tìm ra tên chuỗi của mục tin được chọn.
    MenuItem miClicked = (MenuItem)sender;
    string item = miClicked.Text;

    if(item == "Huge")
    {
        currFontSize = TheFontSize.Huge;
        currentCheckedItem = checkedHuge;
    }
    if(item == "Normal")
    {
        currFontSize = TheFontSize.Normal;
        currentCheckedItem = checkedNormal;
    }
    if(item == "Tiny")
    {
        currFontSize = TheFontSize.Tiny;
        currentCheckedItem = checkedTiny;
    }
    currentCheckedItem.Checked = true;
    Invalidate();
}

```



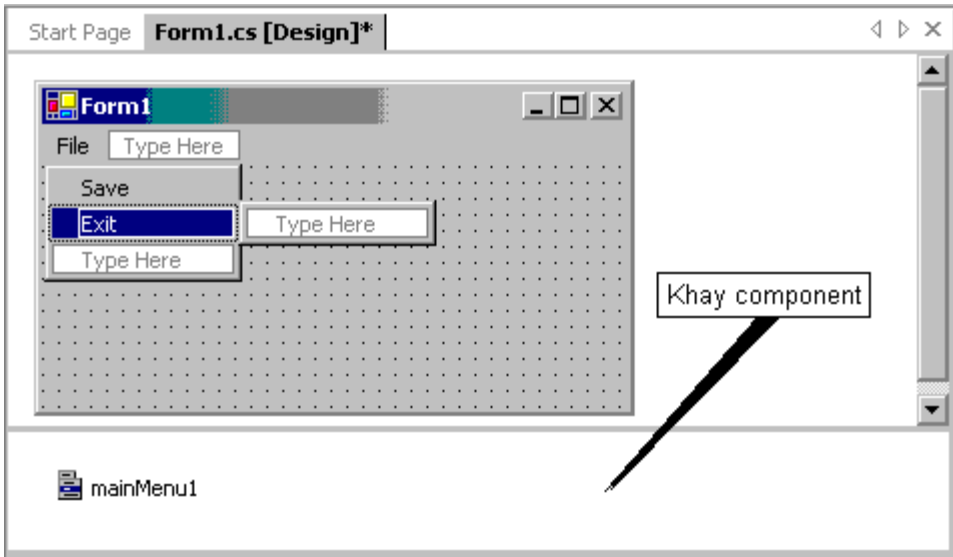
Hình 2-15: Các mục chọn trình đơn có check mark

## 2.6.4 Tạo một trình đơn dùng Visual Studio .NET IDE

Tới đây, bạn đã biết tạo trình đơn thông qua chương trình. Tuy nhiên, Visual Studio .NET IDE cũng cho phép bạn tạo dễ dàng trình đơn vào lúc thiết kế. Muốn thế, trước tiên bạn cho tạo một ứng dụng Windows Form mới, cho hiện lên cửa sổ thiết kế Form1. Bạn cho gọi vào Toolbox (ấn tổ hợp phím <Ctrl+Alt+X>), rồi double-click mục chọn **MainMenu** trên Toolbox.

Lúc này, bạn thấy hiện lên một khung cửa sổ nhỏ ở phía dưới, được gọi là component tray (khay chứa các cấu kiện) có chứa một icon **mainMenu1**. Đồng thời dưới tiêu đề **Form1**, có một khung hình chữ nhật có chữ “Type here”. Đây là chỗ bắt đầu hình thành trình đơn của bạn. Bạn double click vào khung hình chữ nhật rồi gõ chữ “File” chẳng hạn, mục chọn trình đơn chính đầu tiên. Lúc này bạn thấy hiện lên hai ô hình chữ nhật, một bên phải mục “File”, và một ô nằm ngay dưới mục “File”. Bạn chọn ô nằm dưới, double click rồi gõ “Save”. Như vậy “Save” là subitem của “File”. Khi bạn gõ xong “Save”, thì như lúc trước lại hiện lên hai ô trống một bên phải “Save”, và một dưới “Save”. Bạn tiếp tục double click ô nằm dưới “Save”, rồi gõ “Exit”. Lúc này mục chọn “Exit” là subitem của mục chọn “File”. Bạn có hình 2-16.

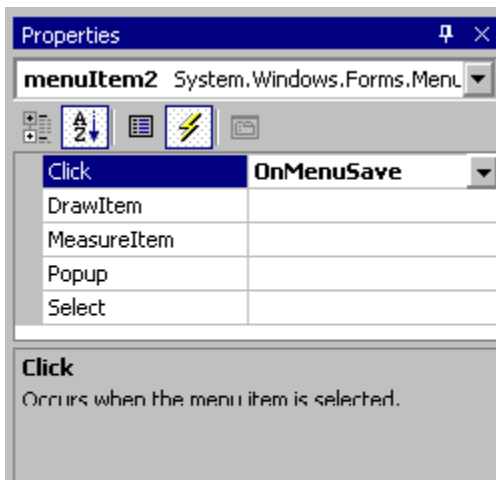
Khi bạn tạo xong trình đơn, bạn có thể bước qua viết các hàm thụ lý tình huống khi mục chọn trình đơn bị click. Bạn cũng có thể dùng cửa sổ **Properties** của trình đơn vừa mới xây dựng để tạo các hàm này.



Hình 2-16: Tạo trình đơn vào lúc thiết kế

Thí dụ bạn click lên mục chọn **Save**, cửa sổ Properties hiện lên chỉ về menuItem2. Bạn click lên ô có dấu “sấm chớp” để cho hiện lên những tình huống của mục chọn trình đơn. Bạn chọn tình huống **Click**, rồi gõ vào tên hàm là **OnMenuSave**. Xem hình 2-17. Rồi bạn ấn <OK>. Lúc này một khung sườn hàm OnMenuSave sẽ hiện lên như sau:

```
private void OnMenuSave(object sender, System.EventArgs e)
{
    // xử lý tình huống Save
}
```



Hình 2-17: Hàm thụ lý tình huống OnMenuSave

Bạn để ý là khi dùng IDE để điều chỉnh trình đơn thì IDE sẽ nhật tu hàm hỗ trợ InitializeComponent(), cũng như thêm những biến thành viên đại diện cho các lớp mà bạn thao tác trong thời gian thiết kế. Nếu bạn bỏ công xem đoạn mã do IDE kết sinh thì bạn thấy giống như đoạn mã bạn tự viết lấy, nhưng với bao nhiêu nhọc nhằn.

## 2.7 Tìm hiểu thanh tình trạng và lớp StatusBar

Ngoài hệ thống trình đơn, nhiều biểu mẫu còn có thanh tình trạng (status bar), thường nằm cuối màn hình. StatusBar control thường dùng hiển thị những thông tin ngắn, không ngắt ngào hoặc là những thông tri không quan trọng. Sau đây là một số thông tin mà thanh tình trạng mang lại:

**Thông tin liên quan đến chế độ ứng dụng hoặc phạm trù hoạt động:** Thí dụ, nếu ứng dụng có thể chạy bởi nhiều người sử dụng khác nhau, có thể bạn cần một thanh tình trạng cung cấp thông tin liên quan đến cấp người sử dụng (user level) hiện hành (chẳng hạn Administrator mode, v.v...). Cũng tương tự như thế, một ứng dụng tài chính có thể cung cấp một label cho biết tỉ giá hối đoái đồng USD chẳng hạn hoặc cho chuyển bật qua lại giữa nhiều tiền tệ khác nhau.

**Thông tin liên quan đến tình trạng ứng dụng:** Thí dụ, một ứng dụng căn cứ dữ liệu có thể bắt đầu hiển thị **Ready** hoặc **Connected To...** khi lần đầu tiên bạn đăng ký vào, rồi sau đó cho hiển thị **Record Added** khi bạn nhập tu căn cứ dữ liệu.

**Thông tin liên quan đến một công việc đang được xử lý ở hậu trường:** Thí dụ, Word cung cấp một vài thông tin liên quan đến việc in ấn trên một thanh tình trạng.

**Thông tin liên quan đến document hiện hành:** Microsoft Word dùng một thanh tình trạng để hiển thị số trang hiện hành và vị trí người sử dụng hiện đang ở trên tài liệu, v.v..

### 2.7.1 Cơ bản về StatusBar

Status bar có thể được chia thành những ô nhỏ được gọi là “pane” hoặc “panel” (khung cửa nhỏ). Những pane này thường hiển thị những thông tin kiểu văn bản hoặc kiểu đồ họa, chẳng hạn dòng văn bản hướng dẫn, hoặc thông tin cụ thể của ứng dụng v.v.. Bạn có hai cách sử dụng đến status bar: cách thứ nhất hiển thị một thanh tình trạng đơn giản, và cách thứ hai là phối hợp những panel chứa “xà ngang” văn bản và icon.

Muốn dùng một thanh tình trạng đơn giản, bạn chỉ cần đặt để thuộc tính **ShowPanels** và **Text**, như sau. Điều này không bao thanh tình trạng bởi một đường viền (xem hình 2-18).

```
statusBar.ShowPanels = false;  
statusBar.Text = "Ready";
```





**Hình 2-18: Thanh tình trạng đơn giản**

đối tượng **StatusBarPanel** trên thuộc tính **Panels**. Bạn có thể cấu hình collection này vào lúc design time hoặc bằng chương trình vào lúc runtime:

```
StatusBarPanel pnlNew = new StatusBarPanel();
pnlNew.Text = "Ready";
statusBar.Panels.Add(pnlNew);
```

Lớp **StatusBar** dẫn xuất trực tiếp từ **System.Windows.Forms.Control**. Ngoài những thành viên được kế thừa, lớp **StatusBar** còn định nghĩa một số thuộc tính cốt lõi như theo bảng 2-20 sau đây:

**Bảng 2-20: Các thuộc tính cốt lõi của lớp *StatusBar***

Các thuộc tính	Ý nghĩa
<b>BackgroundImage</b>	Đi lấy hoặc đặt để hình ảnh được hiện lên trên nền của ô control statusbar.
<b>Font</b>	Đi lấy hoặc đặt để phông chữ mà ô StatusBar control sẽ sử dụng đến để hiển thị thông tin.
<b>ForeColor</b>	Đi lấy hoặc đặt để màu forecolor của ô statusbar control.
<b>Panels</b>	Thuộc tính này trả về một lớp nằm lồng <b>StatusBarPanelCollection</b> chứa mỗi panel duy trì bởi SatusBar (tương tự như bên Menu).
<b>ShowPanels</b>	Đi lấy hoặc đặt để một trị cho biết panels có nên cho hiển thị hay không.
<b>SizingGrip</b>	Đi lấy hoặc đặt để một trị cho biết một sizing grip sẽ hiện lên hay không ở một góc trên ô statusbar control.

Một khi bạn đã tạo xong một đối tượng **StatusBar**, công việc kế tiếp là thêm bất cứ số lượng nào panel (tượng trưng bởi lớp **StatusBarPanel**) vào collection mang tên **StatusBar\$StatusBarPanelCollection**. Bạn để ý là hàm constructor của lớp collection này tự động tạo một panel mới với dáng dấp mặc nhiên. Nếu bạn bằng lòng với trị mặc

Bạn cũng có thể thay đổi phông chữ trên phần văn bản, hoặc đặt để thuộc tính **SizingGrip** cho phép vô hiệu hoá hoặc cho hiệu lực những grip line ở cuối góc phải (nơi mà người sử dụng cần lôi để co dãn biểu mẫu

Cách thứ hai là bạn có thể tạo một thanh tình trạng với nhiều thông tin lẫn lộn khác nhau, được tượng trưng bởi một collection các

nhiên này thì việc lập trình của bạn sẽ đơn giản hơn nhiều. Bảng 2-21 liệt kê các thuộc tính cốt lõi của lớp `StatusBarPanel` với trị mặc nhiên:

**Bảng 2-21: Các thuộc tính cốt lõi của lớp `StatusBarPanel`**

Các thuộc tính	Ý nghĩa
<b>Alignment</b>	Thuộc tính này ấn định cách canh văn bản trên panel. Trị mặc nhiên là <b>HorizontalAlignment</b> – canh theo chiều ngang.
<b>AutoSize</b>	Thuộc tính này ấn định liệu xem panel này tự động điều chỉnh lại kích thước (và bằng cách nào) hay không. Trị mặc nhiên là <b>StatusBarPanel.AutoSize.None</b> .
<b>BorderStyle</b>	Thuộc tính này cấu hình kiểu đường viền của panel. Trị mặc nhiên là <b>StatusBarPanel.BorderStyle.Sunken</b> .
<b>Icon</b>	Thuộc tính này cho biết có icon hay không? Trị mặc nhiên là null, nghĩa là không có icon.
<b>MinWidth</b>	Kích thước tối thiểu của panel. Trị mặc nhiên là 10.
<b>Style</b>	Panel chứa cái gì?. Trị mặc nhiên là <b>StatusBarPanelStyle.Text</b> , nhưng còn có những kiểu style khác được kê khai trong enumeration <b>StatusBarPanelStyle</b> .
<b>Text</b>	Tựa đề (caption) của panel. Trị mặc nhiên là chuỗi chữ rỗng.
<b>ToolTipText</b>	Có tooltip hay không? Chuỗi rỗng là trị mặc nhiên.
<b>Width</b>	Chiều dài của panel. Trị mặc nhiên là 100.

## 2.7.2 Thử tạo một thanh tình trạng

Bây giờ ta thử tạo một đối tượng **StatusBar**, chia nó thành 2 pane. Pane thứ nhất cho biết những câu nhắc (prompt) mô tả chức năng của mỗi mục chọn trình đơn được tuyển. Còn pane thứ hai, nằm ở bên phải, cho hiển thị ngày giờ của hệ thống. Xem hình 2-18 trang 58 ở trên.

Giả sử bạn cho nhật tu ứng dụng **SimpleMenu** mà bạn đã tạo ra trước đó trong chương này, chịu hỗ trợ statusbar này. Giống như mọi lớp được dẫn xuất từ lớp **Control**, đối tượng **StatusBar** cần được thêm vào collection **Controls** của Form. Như bạn có thể đoán được, collection này chứa tất cả các món “lục lãng lục chót” GUI được đưa vào client area, bao gồm đối tượng **StatusBar**. Sau đây là đoạn mã thêm vào liên quan đến thanh tình trạng, và timer mà chúng tôi sẽ xem tiếp trong phần tới:

```
namespace StatusBar
{
    using System;
    using System.Drawing;
    using System.Collections;
    using System.ComponentModel;
    using System.Windows.Forms;
    using System.Data;

    public class MainForm: Form
    {
        // Các biến thành viên dùng cho statusbar và pane
        private StatusBar statusBar = new StatusBar();
        private StatusBarPanel sbPnlPrompt = new StatusBarPanel();
        private StatusBarPanel sbPnlTime = new StatusBarPanel();

        // biến đồng hồ
        private Timer timer1 = new Timer();

        ...

        public MainForm()
        {
            ...
            // Cấu hình timer.
            timer1.Interval = 1000;
            timer1.Enabled = true;
            timer1.Tick += new EventHandler(timer1_Tick);

            BuildStatBar(); // thực hiện mọi việc liên quan
                           // đến statusbar
        }

        ...

        // Hàm thụ lý tình huống timer ...
        private void timer1_Tick(object sender, EventArgs e)
        {
            DateTime t = DateTime.Now;
            string s = t.ToLongTimeString();
            sbPnlTime.Text = s;
        }

        private void BuildStatBar()
        {
            // Cấu hình status bar
            statusBar.ShowPanels = true;
            statusBar.Size = new System.Drawing.Size(212,20);
            statusBar.Location = new System.Drawing.Point(0, 216);

            // AddRange() cho phép bạn thêm một lô pane ngay lập tức
            statusBar.Panels.AddRange(new StatusBarPanel[]
            {sbPnlPrompt, sbPnlTime});
        }
    }
}
```

```

// Cấu hình prompt panel
sbPnlPrompt.BorderStyle = StatusBarPanelBorderStyle.None;
sbPnlPrompt.AutoSize = StatusBarPanelAutoSize.Spring;
sbPnlPrompt.Width = 62;
sbPnlPrompt.Text = "Ready";

// Cấu hình time pane
sbPnlTime.Alignment = HorizontalAlignment.Right;
sbPnlTime.Width = 76;

// Thêm một icon!
try
{
    // icon này phải nằm cùng thư mục ứng dụng
    Icon i = new Icon("status.ico");
    sbPnlPrompt.Icon = i;
}
catch (Exception e)
{
    MessageBox.Show(e.Message);
}

// Bây giờ thêm status bar mới này vào Controls collection.
this.Controls.Add(statusBar);
}
}

```

Kết quả chạy chương trình này là hình 2-18.

**TIP** Có thể bạn nên dùng những biến cấp biểu mẫu (form-level) để trừ một qui chiếu về các đối tượng **StatusBarPanel** mà bạn cần nhậ tu thường xuyên.

## 2.7.3 Đồng bộ hoá thanh tình trạng với một trình đơn

Không có cách nào tự động hoá nổi một văn bản Help của mục chọn trình đơn về một status bar panel. Trước khi chúng tôi chỉ cho bạn cách làm, chúng tôi muốn nhắc nhở bạn không nên làm thế.

Vấn đề đặt một văn bản Help lên thanh tình trạng là kể cả những người sử dụng tiến bộ ít khi gắn liền hai ô control lại với nhau khi chúng cách xa nhau khá dài về mặt vật lý. Nói cách khác, ai cần Help text sẽ không biết là nó có ở đấy. Phần lớn người sử dụng không hiểu là họ có thể lớn vồn trên một mục chọn trình đơn để chọn nó, và có thể họ chỉ hiểu là click lên mục chọn trình đơn sẽ kích hoạt lập tức một hành động nào đó mà không cần cung cấp thông tin giúp đỡ. Nhiều người sử dụng than phiền văn bản Help trên thanh tình trạng chỉ tổ tốn chỗ mà thôi.

Tuy nhiên, nếu bạn cứ khẳng định tạo giao diện này, thì bạn chỉ cần dùng hàm hành sự **MenuItem.Select()**. Lớp **MenuItem** không cung cấp bất cứ thuộc tính **Tag** nào mà bạn có thể trữ thông tin bổ sung (chẳng hạn Help text). Thay vào đó, bạn chuẩn bị sẵn một hashtable collection, cho mang tên HelpCollection chẳng hạn, và dùng qui chiếu sender về ô control để dò tìm chuỗi văn bản Help.

```
// hàm này thụ lý mnuOpen.Click, mnuNew.Click, mnuSave.Click, v.v...
private void mnu_Click(object sender, EventArgs e)
{
    Status.Text = HelpCollection[sender];
}
```

Bạn cũng có thể làm được điều này bằng cách dùng một custom menu control rồi cho phủ quyết (override) hàm hành sự **OnSelect()**.

## 2.7.4 Làm việc với lớp Timer

Pane thứ hai, nằm ở bên phải trên hình 2-18, sẽ cho hiển thị một đồng hồ hệ thống. Đối tượng Windows Form **Timer** là một lớp khá đơn giản gồm một số hàm hành sự mà chúng tôi cho liệt kê sau đây, bảng 2-22:

**Bảng 2-22: Các thành viên của lớp Timer**

Các thành viên	Ý nghĩa
<b>Enabled</b>	Thuộc tính này cho hiệu lực hoặc vô hiệu hoá khả năng đối tượng <b>Timer</b> cho phát ra tình huống <b>Tick</b> . Bạn cũng có thể dùng <b>Start()</b> và <b>Stop()</b> để đưa đến cùng kết quả.
<b>Interval</b>	Thuộc tính này cho đặt để khoảng cách thời gian giữa hai tick. Tính theo millisecond.
<b>Start()</b>	Giống như thuộc tính <b>Enabled</b> , các hàm hành sự này điều khiển việc phát ra tình huống <b>Tick</b> .
<b>Stop()</b>	
<b>OnTick()</b>	Hàm này có thể bị phủ quyết trên một custom class được dẫn xuất từ lớp <b>Timer</b> .
<b>Tick event</b>	Tình huống <b>Tick</b> thêm một thụ lý tình huống mới vào <b>Multicast Delegate</b> nằm đăng sau.

Đoạn mã liên quan đến timer đã được liệt kê trong thí dụ trên. Bạn để ý, hàm thụ lý tình huống timer1\_Tick sử dụng đến kiểu dữ liệu DateTime. Bạn có thể tìm thấy thời gian hệ thống hiện hành bằng cách sử dụng đến thuộc tính **Now** để đặt để thuộc tính **Text** của đối tượng **StatusBarPanel** thích ứng. Bạn xem lại đoạn mã thí dụ kể trên.

## 2.7.5 Cho hiển thị dòng nhắc đối với mục chọn trình đơn

Cuối cùng bạn phải lập trình cho pane thứ nhất dùng hiển thị những dòng nhắc (prompt) khi một mục chọn trình đơn được tuyển. Ta tiếp tục lấy thí dụ ứng dụng trình đơn đơn giản kể trên. Tuy nhiên, lần này ta đáp ứng tình huống **Select** khi mỗi subitem được tuyển. Khi người sử dụng chọn “File | Exit” hoặc “Help | About” thì bạn yêu cầu đối tượng đầu tiên của StatusBarPanel phải cho hiển thị một thông điệp văn bản nào đó. Ngoài ra, bạn cũng phải thụ lý tình huống **MenuComplete** bảo đảm rằng khi người sử dụng kết thúc thao tác trình đơn thì cho thông điệp mặc nhiên lên pane thứ nhất của thanh tình trạng. Sau đây là đoạn mã bổ sung:

```
public class MainForm: Form
{
    ...
    public MainForm()
    {
        ...
        // Tình huống MenuComplete được phát ra khi người sử dụng click
        // khỏi menu. Ta cần chặn hứng tình huống này để có thể đặt
        // chuỗi "Ready" lên pane thứ nhất của status bar. Nếu không
        // làm thế thì văn bản StatusBarPanel bao giờ cũng dựa vào
        // trình đơn chót được tuyển.
        this.MenuComplete += new EventHandler(StatusForm_MenuDone);
        BuildMenuSystem();
    }

    // Hàm thụ lý tình huống click menu File | Exit
    private void FileExit_Clicked(object sender, EventArgs e)
    {
        this.Close();
    }

    // Hàm thụ lý tình huống click menu Help | About
    private void HelpAbout_Clicked(object sender, EventArgs e)
    {
        MessageBox.Show("Trình đơn kỳ lạ...");
    }

    // Hàm thụ lý tình huống menu File | Exit được tuyển.
    private void FileExit_Selected(object sender, EventArgs e)
    {
        sbPnlPrompt.Text = "Chấm dứt ứng dụng này";
    }

    // Hàm thụ lý tình huống menu Help | About được tuyển.
    private void HelpAbout_Selected(object sender, EventArgs e)
    {
        sbPnlPrompt.Text = "Hiển thị thông tin ứng dụng ";
    }

    // Hàm thụ lý tình huống khác ...
}
```

```

private void StatusForm MenuDone(object sender, EventArgs e)
{
    sbPnlPrompt.Text = "Ready";
}

// Các hàm hỗ trợ .
private void BuildMenuSystem()
{
    // Trước tiên tạo main menu
    mainMenu = new MainMenu();

    // Tạo 'File' Menu.
    MenuItem miFile = mainMenu.MenuItems.Add("&File");
    miFile.MenuItems.Add(new MenuItem("E&xit",
        new EventHandler(this.FileExit_Clicked),
        Shortcut.CtrlX));

    // thụ lý tình huống Select đối với mục chọn trình đơn Exit
    miFile.MenuItems[0].Select += new
        EventHandler(FileExit_Selected);

    // Bây giờ tạo 'Help | About' menu.
    MenuItem miHelp = mainMenu.MenuItems.Add("Help");
    miHelp.MenuItems.Add(new MenuItem("&About",
        new EventHandler(this.HelpAbout_Clicked),
        Shortcut.CtrlA));

    // thụ lý tình huống Select đối với mục chọn trình đơn About
    miHelp.MenuItems[0].Select += new
        EventHandler(HelpAbout_Selected);

    // Ghép main menu về đối tượng Form
    this.Menu = mainMenu;
}
...
}

```

Tới đây coi như mọi việc đều tốt lành. Visual Studio .NET IDE cũng cung cấp một số hỗ trợ khi thiết kế các đối tượng status bar. Bạn sẽ học cách tạo một thanh công cụ (toolbar) sử dụng đến IDE. Một khi bạn hiểu được tiến trình tạo một toolbar, thì bạn cũng có thể tạo một status bar sử dụng đến những công cụ vào lúc thiết kế.

## 2.8 Tạo một thanh công cụ

Lớp cuối cùng mà ta sẽ xét đến là lớp **ToolBar**. Ô control **ToolBar** tượng trưng cho một dải nút cho phép bạn truy cập những chức năng khác nhau trên một ứng dụng. Như bạn đã biết thanh công cụ là công cụ thay thế việc phải hiện dịch một mục chọn trình đơn nào đó. Do đó, khi người sử dụng click lên nút “Save” có thể xem như người sử dụng ra lệnh “File | Save”. Về mặt khái niệm, thanh công cụ giữ vai trò tương tự như trình đơn, nhưng có chút khác biệt là thanh công cụ chỉ sâu một tầng và thường dành cho những chức năng quan trọng nhất thường xuyên được dùng đến. Còn trình đơn sử dụng đến một

kiến trúc nhiều tầng đem lại một điểm truy cập duy nhất vào tất cả các chức năng của ứng dụng.

Thanh công cụ có thể bao gồm văn bản và hình ảnh chiều dài thay đổi. Bạn có thể tạo nhiều tầng lớp thanh công cụ trên một biểu mẫu cũng như gắn một trình đơn kéo xuống (drop-down menu) lên một nút nào đó.

Trên namespace `Windows.Forms` có một số lớp cho phép bạn tạo những thanh công cụ như thế. Ta thử bắt đầu xét đến lớp **ToolBar**. Bảng 2-23 liệt kê các thuộc tính của lớp này:

**Bảng 2-23: Các thuộc tính cốt lõi của lớp *ToolBar***

Các thuộc tính	Ý nghĩa
<b>Appearance</b>	<b>ToolBarAppearance.Normal</b> làm cho nút xuất hiện theo kiểu 3 chiều. <b>ToolBarAppearance.Flat</b> thì làm cho nút xuất hiện chẹt bẹt (flat) nhưng khi con nháy chuột “rờ” lên nó thì nó nhô lên.
<b>AutoSize</b>	Nếu thuộc tính này về true (trị mặc nhiên) thì thanh công cụ tự điều chỉnh dựa trên kích thước nút, số nút và thuộc tính <b>DockStyle</b> của thanh công cụ.
<b>BorderStyle</b>	Thuộc tính này cho biết loại đường viền nào bao quanh ô control như được khai báo bởi enumeration <b>BorderStyle</b> . Khi cho về <b>BorderStyle.Fixed3D</b> thì thanh công cụ mang dáng 3 chiều, còn khi cho về <b>BorderStyle.FixedSingle</b> thì thanh công cụ có một đường viền lép mỏng dính.
<b>Buttons</b>	Đây là một collection thuộc quyền thanh công cụ (nghĩa là <b>ToolBar\$ToolBarButtonCollection</b> )
<b>ButtonSize</b>	Thuộc tính này ấn định kích thước của một button trên thanh công cụ. Trị mặc nhiên là 24x22 pixels.
<b>DropDownArrows</b>	Nếu thuộc tính này là false, thì sẽ không có mũi tên chúc xuống đối với thanh công cụ có gắn kết với trình đơn (mặc dù trình đơn kéo xuống vẫn hiện lên khi ta click lên nút). Khi thuộc tính này về true, thì nút sẽ kèm theo một mũi tên chúc xuống cho phép người sử dụng click lên mũi tên để cho hiện lên trình đơn.
<b>ImageList</b>	Thuộc tính này trả về ô control <b>ImageList</b> duy trì những hình ảnh của thanh công cụ này.
<b>ImageSize</b>	Đi lấy kích thước của những hình ảnh trên <b>ImageList</b> của thanh công cụ.
<b>ShowToolTips</b>	Thuộc tính này cho biết liệu xem <b>ToolBar</b> sẽ cho hiển thị hay không tooltip đối với mỗi button. Nếu true, thì bạn có thể đặt để thuộc tính <b>ToolTipText</b> của mỗi nút để gắn một tooltip.



**Wrappable**

Các nút thanh công cụ có thể tùy chọn “wrap” qua hàng kế tiếp khi thanh công cụ quá chật hẹp để có thể cho bao gồm tất cả các nút lên cùng một hàng.

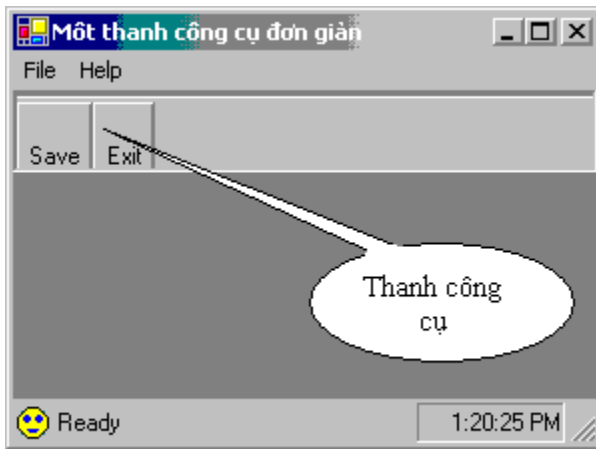
Trong một đối tượng **ToolBar** phần quan trọng là thuộc tính **Buttons**, một collection chứa tập hợp những nút control. Bạn có thể thêm từng đối tượng **ToolBarButton** vào collection này thông qua Visual Studio .NET IDE hoặc bằng chương trình.

Khi một biểu mẫu duy trì một (hoặc hai) **ToolBar** mục tiêu là tạo ra một vài đối tượng riêng rẽ của lớp **ToolBarButton**, tương trưng cho một button, rồi đưa chúng vào lớp collection **ToolBar\$ToolBarButtonCollection**. Mỗi button có thể chứa văn bản, hình ảnh hoặc cả hai. Để cho đơn giản, ta thử tạo một thanh công cụ chứa hai button chỉ cho hiển thị text prompt. Bảng 2-24 cho liệt kê một số thành viên quan trọng của lớp **ToolBarButton**.

**Bảng 2-24: Các thuộc tính cốt lõi của lớp ToolBarButton**

Các thuộc tính	Ý nghĩa
<b>DropDownMenu</b>	ToolBarButtons có thể tùy chọn khai báo một trình đơn popup được hiển thị bất cứ lúc nào một drop-down button bị ấn xuống. Thuộc tính này cho phép bạn kiểm soát việc trình đơn nào sẽ được hiển thị. Và chỉ xuất hiện nếu thuộc tính <b>Style</b> được cho về <b>DropDownButton</b> .
<b>Enabled Visible</b>	Khi bị vô hiệu hóa (not enabled) button sẽ hiện lên mờ cảm, và không phản ứng trước việc button bị click. Button nào không Visible sẽ không hiện lên trên thanh công cụ.
<b>ImageIndex</b>	Thuộc tính này trả về chỉ mục của hình ảnh mà ToolBarButton này đang sử dụng đến. Chỉ mục được đưa từ <b>ImageList</b> của ToolBar cha-mẹ
<b>PartialPush Pushed</b>	Bạn có thể cho <b>Pushed</b> về true để indent một nút. Đây là loại nút kiểu công tắc điện bật qua bật lại được (toggle button). PartialPush cho thấy một nút mờ mờ (dimmed) bị ấn xuống, cho thấy một phối hợp tình trạng pushed và unpushed.
<b>Style</b>	Thuộc tính này trả về style của toolbar button. Nó sẽ hình thành enumeration <b>ToolBarButtonStyle</b> . Enum này như sau: <b>DropDownButton</b> Cho biết một ô control kéo xuống sẽ hiển thị một trình đơn hoặc một khung đối thoại khác khi nút bị click. <b>PushButton</b> Một nút chuẩn, 3 chiều <b>Separator</b> Một space hoặc một hàng phân cách giữa các nút, tùy thuộc vào trị của thuộc tính Appearance

	<b>ToggleButton</b>	Một nút kiểu công tắc điện mang dáng dấp sụm xuống khi bị click và ở tình trạng này cho tới khi bị click lại.
<b>Text</b>		Tựa đề (caption) sẽ hiển thị trên ToolBar button.
<b>ToolTipText</b>		Nếu ToolBar cha-mẹ có thuộc tính <b>ShowToolTips</b> là true, thì thuộc tính này mô tả dòng văn bản sẽ được hiển thị trên button.
<b>Visible</b>		Thuộc tính này cho biết liệu xem button có hiện lên hay không. Nếu không thì nó sẽ không hiện lên và sẽ không có khả năng nhận dữ liệu nhập.



Thanh công cụ tự tạo sẽ có hai nút “Save” và “Exit” (hình 19), phản ánh hành vi được cung cấp bởi mục chọn trình đơn Save và Exit. Sau đây là đoạn mã thêm vào thí dụ status bar ở trên:

Hình 2-19: Một thanh công cụ đơn giản

```
public class MainForm: Form
{
    // Dữ liệu tình trạng đối với toolbar và hai button
    private ToolBarButton tbSaveButton = new ToolBarButton();
    private ToolBarButton tbExitButton = new ToolBarButton();
    private ToolBar toolBar = new ToolBar();

    public MainForm()
    {
        ...
        // Dân dụng những thuộc tính của Form
        Text = "Một thanh công cụ đơn giản";
        CenterToScreen();
        BackColor = Color.CadetBlue;
        ...
        BuildToolBar();
    }
    ...

    private void BuildToolBar()
    {
        // Cấu hình từng button
```

```

tbSaveButton.Text = "Save";
tbSaveButton.ToolTipText = "Save";
tbExitButton.Text = "Exit";
tbExitButton.ToolTipText = "Exit";

// Cấu hình ToolBar và thêm button vào
toolBar.BorderStyle = System.Windows.Forms.BorderStyle.Fixed3D;
toolBar.ShowToolTips = true;
toolBar.Buttons.AddRange(new ToolBarButton[] {tbSaveButton,
                                                tbExitButton});

toolBar.ButtonClick += new
    ToolBarButtonClickEventHandler(ToolBar_Clicked);

// thêm thanh công cụ mới vào collection Controls
this.Controls.Add(toolBar);
}

// Hàm thụ lý tình huống click ToolBar
private void ToolBar_Clicked(object sender,
    ToolBarButtonClickEventArgs e)
{
    MessageBox.Show(e.Button.ToolTipText);
}

```

Hình 19 cho thấy kết quả chương trình chạy lại:

Trong chốc lát, chúng tôi sẽ thêm vài hình ảnh. Trong khi chờ đợi, ta thử phân tích đoạn mã kể trên. Hàm hỗ trợ BuildToolBar() bắt đầu cấu hình hoá một vài thuộc tính cơ bản đối với mỗi đối tượng ToolBarButton. Kế tiếp, đưa chúng vào ToolBar collection sử dụng hàm hành sự AddRange(), thay vì triệu gọi hàm hành sự Add() nhiều lần. Muốn thụ lý tình huống Click đối với một nút nào đó, bạn phải thụ lý tình huống ButtonClick:

```

toolBar.ButtonClick += new
    ToolBarButtonClickEventHandler(ToolBar_Clicked);

```

Tên của delegate **ToolBarBarButtonClickEventHandler** phải có một dấu ấn theo dãy thông số thứ hai thuộc kiểu dữ liệu **ToolBarBarButtonClickEventArgs**. Kiểu dữ liệu này có thể được xem xét để xác định nút nào phát đi tình huống, sử dụng thuộc tính **Button**.

```

// Hàm thụ lý tình huống click ToolBar
private void ToolBar_Clicked(object sender,
    ToolBarButtonClickEventArgs e)
{
    // Chỉ cho thấy toolbar text tương ứng
    MessageBox.Show(e.Button.ToolTipText);
}

```

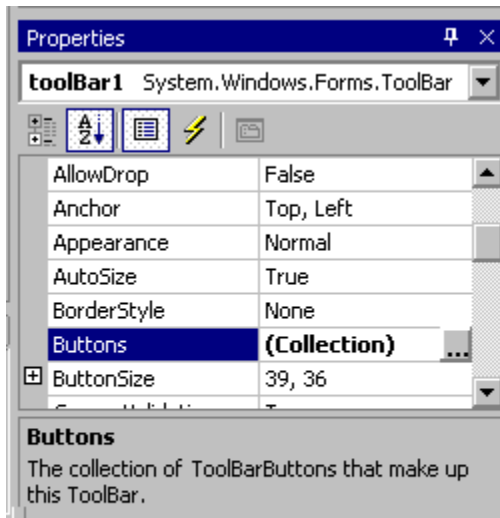


```

        toolBar.ButtonClick += new ToolBarButtonClickEventHandler(
            ToolBar_Clicked);

    // Nạp hình ảnh (*.ico phải nằm cùng thư mục ứng dụng)
    toolBarIcons.ImageSize = new System.Drawing.Size(32, 32);
    toolBarIcons.Images.Add(new Icon("filesave.ico"));
    toolBarIcons.Images.Add(new Icon("fileexit.ico"));
    toolBarIcons.ColorDepth = ColorDepth.Depth16Bit;
    toolBarIcons.TransparentColor = System.Drawing.Color.Transparent;
    this.Controls.Add(toolBar);
}
... }

```



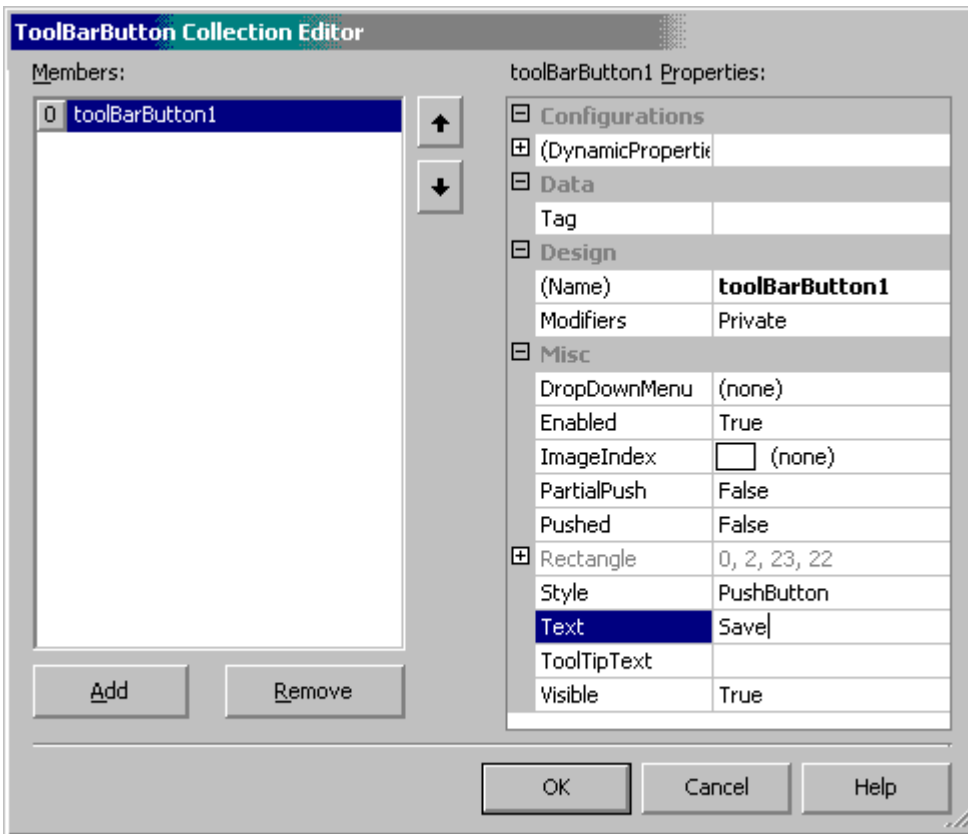
**Hình 2-21: Thêm các nút ToolStrip vào lúc thiết kế.**

Nếu bạn cho chạy lại ứng dụng sau khi đã nhập tu, kết quả như mong muốn. Xem hình 2-20.

## 2.8.2 Tạo thanh công cụ vào lúc thiết kế

Tạo thanh công cụ vào lúc thiết kế có thể thực hiện bằng cách sử dụng đến cửa sổ **Properties**. Thí dụ, nếu bạn muốn thêm những nút Button vào đối tượng ToolStrip, bạn double-click lên thuộc tính **Button** trên cửa sổ **Properties** (hình 2-21).

Lúc này một khung đối thoại **ToolStrip Button Collection Editor** sẽ hiện lên (hình 2-22). Khung đối thoại này cho phép bạn thêm, gỡ bỏ hoặc cấu hình riêng rẽ từng mục **ToolStripButton**.

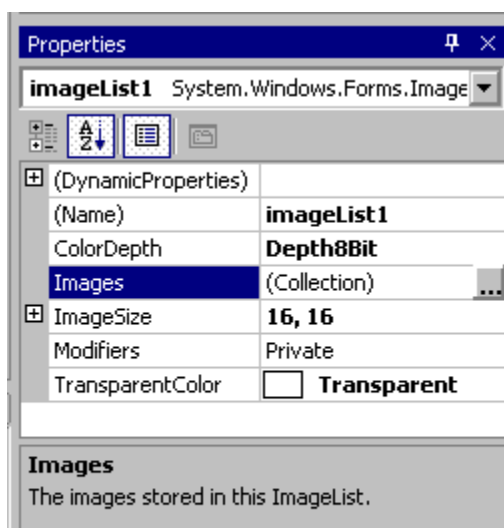


Hình 2-22: Cấu hình nút trên thanh công cụ vào lúc thiết kế

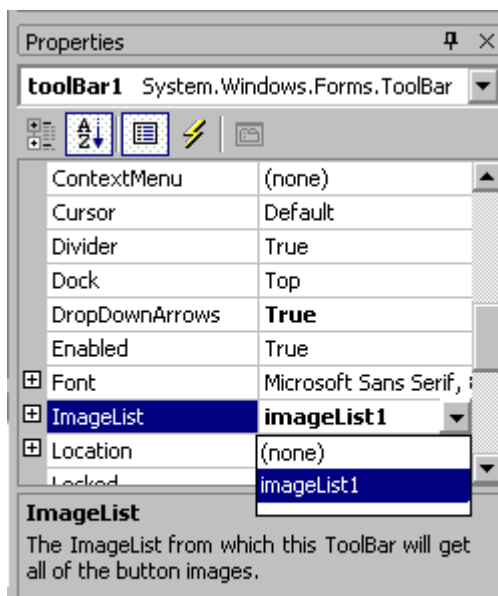
### 2.8.2.1 Thêm một *ImageList* vào lúc thiết kế

Bạn để ý là cũng khung đối thoại (hình 2-22) kể trên cho phép bạn gán một hình ảnh icon cho mỗi button sử dụng đến thuộc tính **ImageIndex**. Tuy nhiên, thuộc tính này vô dụng cho tới khi bạn thêm một đối tượng **ImageList** lên dự án hiện hành của bạn. Muốn thêm một đối tượng **ImageList** lên biểu mẫu vào lúc thiết kế, bạn gọi vào **Toolbox** rồi ấn lên mục **ImageList**. Lúc này đối tượng **ImageList** (imageList1) nằm trên Component Tray, một cửa sổ nhỏ ở dưới màn hình dùng chứa các component (cấu kiện).

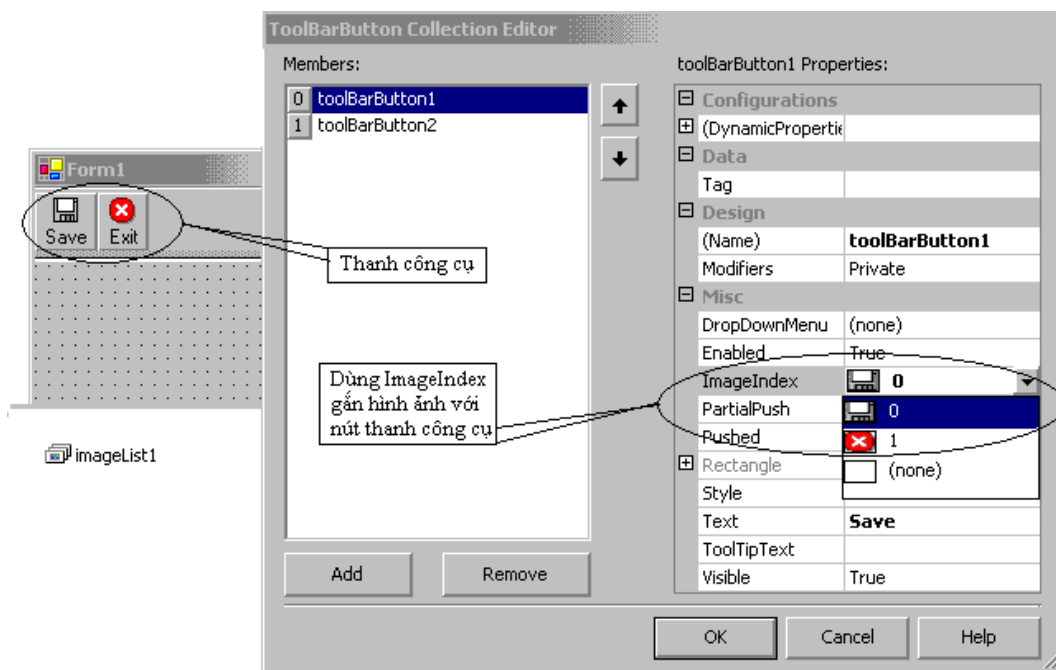
Tới đây, bạn có thể dùng cửa sổ **Properties** để thêm những hình ảnh riêng rẽ mà thuộc tính **Images** sẽ dùng đến. Xem hình 2-23. Một khi bạn đã thêm mỗi hình ảnh vào **ImageList**, bạn thông báo cho ToolBar biết **ImageList** nào sẽ dùng đến, cũng thông qua cửa sổ **Properties** (hình 2-24).



Hình 2-23: Thêm hình ảnh vào ImageList



Hình 2-24: Gắn kết một ImageList với ToolBar



Hình 2-25: Gép hình ảnh với nút thanh công cụ

Tới đây, ta trở về **ToolBarButton Collection Editor** (hình 2-22), bạn click lên mục **ImageIndex** cho hiện lên danh sách các hình ảnh được đánh số từ 0 đến N. Lúc này bạn gán hình ảnh nào đó cho một nút nào đó trên thanh công cụ (xem hình 2-25).

### 2.8.3 Đồng bộ hóa thanh công cụ

Thông thường, chức năng thanh công cụ thường trùng lặp với một số chức năng trên trình đơn. Như vậy sẽ khó khăn trong việc quản lý các đoạn mã liên quan đến trình đơn và thanh công cụ. Bạn phải cẩn thận vô hiệu hoá hoặc cho hiệu lực các mục chọn trình đơn và các nút thanh công cụ tương ứng. Để đơn giản hoá vấn đề, bạn có thể tạo một lớp trình đơn customized tự động chuyển trạng thái của trình đơn qua cho một ô control thanh công cụ được gắn kết.

Dự án custom control này hơi khó. Trước tiên, nó giới hạn việc bạn sử dụng đến Visual Studio .NET IDE để thiết kế trình đơn, bởi vì các mục chọn trình đơn custom phải được tạo và thêm vào thông qua chương trình.

Cũng có một cách khác tiếp cận vấn đề là tạo một lớp MenuItem custom có trừ một qui chiếu về nút thanh công cụ được gắn kết.

```
public class LinkedMenuItem: MenuItem
{
    public ToolBarButton LinkedButton;

    // Để tiết kiệm chỗ chỉ một hàm constructor nguyên thủy được dùng
    public LinkedMenuItem(string text): base (text)    {}

    public bool Enabled
    {
        get{return base.Enabled;}
        set
        {   base.Enabled = value;
            if (LinkedButton != null)
            {   LinkedButton.Enabled = value;
            }
        }
    }
}
```

Điểm bất lợi của kỹ thuật này là nó đòi hỏi bạn phải kèm cặp thuộc tính **Enabled**, vì nó không thể phủ quyết, và đây là một thủ thuật có thể ngăn trở việc hỗ trợ vào lúc thiết kế đối với trình đơn. Nhưng mặt khác, đây là cách tiếp cận rất uyển chuyển. Bạn cũng có thể thay thế LinkedButton bởi một collection cho phép một loạt ô control được tự động cho hiệu lực hoặc vô hiệu hoá trong thủ tục thuộc tính **Enabled**. Bạn để ý là kiểu lập trình đề phòng thường cho trải nghiệm thuộc tính **LinkedButton** trước khi cố gắng cấu hình ô control được qui chiếu, nếu trong trường hợp chưa làm điều này.



Bạn có thể tạo một linked menu sử dụng đến một lớp như sau:

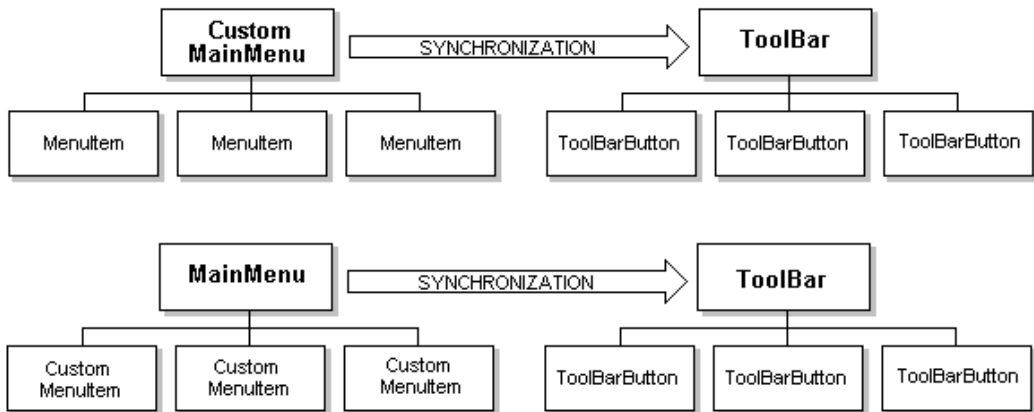
```
MaimMenu menuMain = new MainMenu();
this.Menu = menuMain;

LinkedMenuItem menuItem1 = new LinkedMenuItem("File")'
LinkedMenuItem menuItem2 = new LinkedMenuItem("Exit")'

// ToolBarButton này được định nghĩa như là biến form-level
menuItem2.LinkBarButton = this.ToolBarButton2;

menuMain.MenuItems.Add(menuItem1);
menuItem1.MenuItems.Add(menuItem2);

// Cả hai ToolBarButton và MenuItem bị vô hiệu hoá ngay lập tức
menuItem2.Enabled = false;
```



Hình 2-26: Hai cách đồng bộ hoá một menu và toolbar.

Một cách tiếp cận khác là tạo một lớp MainMenu đặc biệt. Lớp này có thể cung cấp một hàm hành sự vô hiệu hoá các ô control cặp đôi, không phải ép bạn tuân thủ cách làm này nếu bạn muốn làm việc với những đối tượng MenuItem theo ý mình. Điểm bất lợi là bạn có thể quên dùng những hàm hành sự thích ứng và đi đến một tình trạng vô đồng bộ. Ngược lại, cách tiếp cận này đem lại cho bạn nhiều tự do hơn. Hình 2-26 minh họa sự khác biệt:

Việc kết nối được thực hiện đơn giản bằng cách đặt thuộc tính **Tag** của **ToolBarItem** cho qui chiếu về **MenuItem** thích ứng, nhưng cũng có những khả năng khác. **LinkedMainMenu** cần phải rảo lặp (iterate) trên tất cả các button để tìm ra một button khớp.

```

public class LinkedMainMenu: MainMenu
{
    public ToolBar LinkedToolBar;

    public void Enable(MenuItem item)
    {
        SetEnabled(true, item);
    }

    public void Disable(MenuItem item)
    {
        SetEnabled(false, item);
    }

    private void SetEnabled(bool state, MenuItem item)
    {
        item.Enabled = state;
        if (LinkedToolBar != null)
        {
            foreach (ToolBarButton button in LinkedToolBar.Buttons)
            {
                if ((MenuItem)button.Tag == item)
                {
                    button.Enabled = state;
                }
            }
        }
    }
}

```

Và khách hàng có thể theo pattern này:

```

LinkedMaimMenu menuMain = new LinkedMainMenu();
menuMain.LinkedTextBox = myToolBar; // myToolBar là một biến
                                     // form-level
this.Menu = menuMain;

MenuItem menuItem1 = new MenuItem("File");
MenuItem menuItem2 = new MenuItem("Exit");

menuMain.MenuItems.Add(menuItem1);
menuItem1.MenuItems.Add(menuItem2);

// Cả hai ToolBarButton và MenuItem bị vô hiệu hoá ngay lập tức
menuMain.Disable(menuItem2);

```

Trong trường hợp này, ta đã xem qua hai tiếp cận khác nhau để tạo một custom control. Bạn nên nghĩ đến việc tăng cường, customizing và integrating các ô control. Đây là khía cạnh hấp dẫn của lập trình trên .NET.

## 2.9 Một ứng dụng Windows Forms tối thiểu và trọn vẹn

Tới đây bạn có thể tạo một biểu mẫu có đi kèm một trình đơn chính, một trình đơn popup (còn gọi là shortcut menu), một thanh công cụ và một thanh tình trạng. Chương này sẽ kết thúc bằng cách tận dụng những hiểu biết căn bản của bạn về Windows Forms để xây dựng một ứng dụng cuối cùng tóm lược tất cả những gì bạn đã học đến đây.

Ta thử mở rộng chức năng của ứng dụng thanh tình trạng mà ta đã tạo ra trước đó. Bổ sung vào logic hiện hành, ta thử thêm đoạn mã cho đọc và viết dữ liệu ứng dụng lên registry hệ thống cũng như minh họa cách tương tác với Windows 2000 event log.

Trước tiên, ta thử tạo một mục chọn trình đơn chớp bu (“Background Color”) cho phép người sử dụng chọn màu làm nền (background) cho client area từ một lô lựa chọn. Mỗi trình đơn con về màu sắc sẽ có một chuỗi help được gắn liền cần được hiển thị trên pane thứ nhất của đối tượng **StatusBar**. Tình huống **Clicked** đối với mỗi Color subitem sẽ được thụ lý bởi cùng một event handler, **ColorItem\_Clicked**. Tương tự như thế, tình huống **Selected** đối với mỗi subitem sẽ được thụ lý bởi một hàm hành sự mang tên **ColorItem\_Selected**. Sau đây là đoạn mã phải bổ sung:

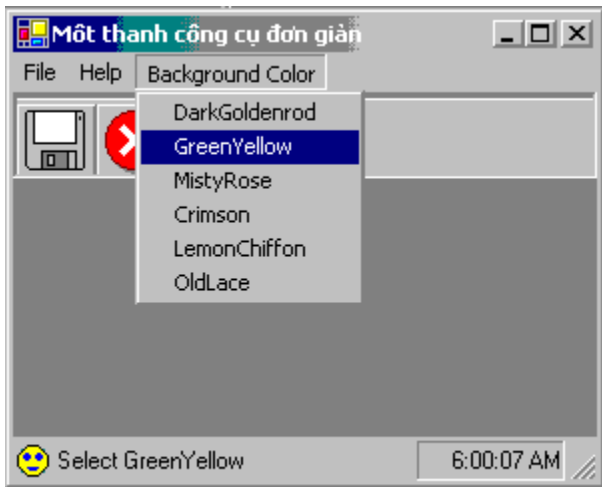
```
private void BuildMenuSystem()
{
    ...
    // Tạo trình đơn "Background Color"
    MenuItem miColor = mainMenu.MenuItems.Add("&Background Color");
    miColor.MenuItems.Add("&DarkGoldenrod",
        new EventHandler(ColorItem_Clicked));
    miColor.MenuItems.Add("&GreenYellow",
        new EventHandler(ColorItem_Clicked));
    miColor.MenuItems.Add("&MistyRose",
        new EventHandler(ColorItem_Clicked));
    miColor.MenuItems.Add("&Crimson",
        new EventHandler(ColorItem_Clicked));
    miColor.MenuItems.Add("&LemonChiffon",
        new EventHandler(ColorItem_Clicked));
    miColor.MenuItems.Add("&OldLace",
        new EventHandler(ColorItem_Clicked));

    // Tất cả các color menu item đều cùng một hàm thụ lý tình huống
    for (int i = 0; i < miColor.MenuItems.Count; i++)
        miColor.MenuItems[i].Select +=
            new EventHandler(ColorMenuItem_Selected);
}
```

Khi người sử dụng chọn ra một subitem nào đó từ trình đơn Background Color, tình huống Select xảy ra. Trong hàm thụ lý tình huống nhiệm vụ của bạn là trích tên văn bản của mục chọn trình đơn được tuyển (nghĩa là OldLace, GreenYellow, chẳng hạn) và cho hiển thị lên thanh tình trạng. Sau đây là đoạn mã:

```
private void ColorMenuItem_Selected(object sender, EventArgs e)
{
    // Cho ra tên chuỗi của item được chọn và gạt bỏ &
    MenuItem miClicked = (MenuItem)sender;
    string item = miClicked.Text.Remove(0,1);

    // giả sử một điểm dữ liệu mới: StatusBarPanell sbPnlPrompt
    sbPnlPrompt.Text = "Select " + item;
}
```



Khi người sử dụng click một color menu item nào đó, bạn chỉ cần cho BackColor của biểu mẫu về màu sắc dựa trên thuộc tính Text của MenuItem. Bạn để ý là chúng tôi làm cho nhớ màu này bằng cách cho trữ trị trên một biến chuỗi mang tên currColor:

**Hình 2-27: Trình đơn Background Color**

```
// hàm thụ lý tình huống Color | X bị click
private void ColorMenuItem_Selected(object sender, EventArgs e)
{
    // Cho ra tên chuỗi của color item được chọn và gạt bỏ &
    MenuItem miClicked = (MenuItem)sender;
    string color = miClicked.Text.Remove(0,1);

    // Bây giờ cho đặt để màu sắc
    this.BackColor = Color.FromName(color);
    Color currColor = BackColor;
}
```

Bạn thử cho chạy lại chương trình, kết quả là hình 2-27.

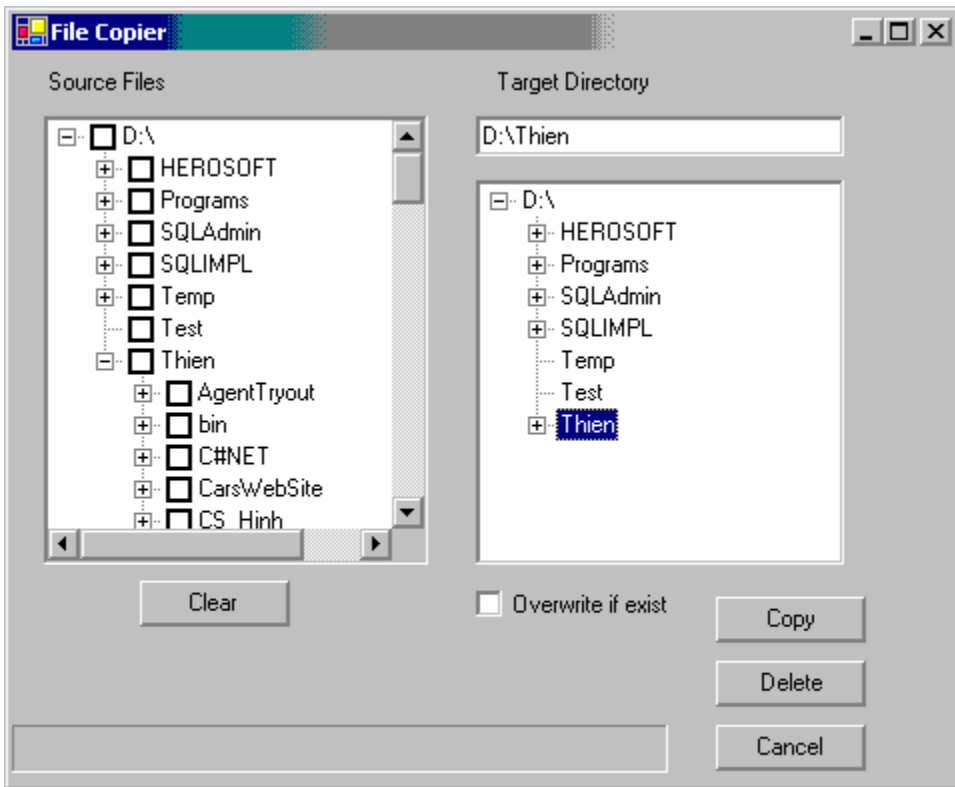
## 2.10 Xây dựng một ứng dụng Windows Form trọn vẹn

Bây giờ để xem **Windows Forms** có thể được dùng thế nào để tạo ra một ứng dụng Windows thực tế hơn, ta thử xây dựng một trình tiện ích mang tên **FileCopier** cho phép bạn chép tất cả các tập tin từ một nhóm thư mục được tuyển chọn bởi người sử dụng qua một thư mục mục tiêu duy nhất hoặc lên một thiết bị, chẳng hạn một đĩa mềm hoặc một đĩa cứng phòng hồ trên mạng của công ty. Mặc dù bạn không thi công mọi tính năng có thể có được, bạn có thể hình dung lập trình ứng dụng này làm thế nào bạn có thể đánh dấu vài chục tập tin rồi cho chép lên nhiều đĩa, cho dồn chặt chứng nào hay chứng nấy. Mục tiêu của thí dụ này là bạn cho thực hành những điều bạn đã học qua trong các phần đi trước cũng như khảo sát namespace Windows Forms.

Bạn tập trung vào giao diện người sử dụng và những bước giúp nối các ô control lại với nhau. Giao diện UI cuối cùng là hình 2-28.

Giao diện UI đối với dự án FileCopier bao gồm các ô control như sau:

- Các ô Labels: Source Files (lblSource) và Target Directory (lblTargetDir), Status (lblStatus).
- Các nút Buttons: Clear (btnClear), Copy (btnCopy), Delete (btnDelete) và Cancel (btnCancel).
- Ô control Checkbox: “Overwrite if exists” (chkOverwrite)
- Ô control Text Box: (txtTargerDir) cho hiển thị path của thư mục đích được chọn ra
- Ô TreeView control: một dành cho các thư mục nguồn có sẵn (twvSource), và một dành cho các thư mục hoặc thiết bị đích có sẵn (twvTargetDir).

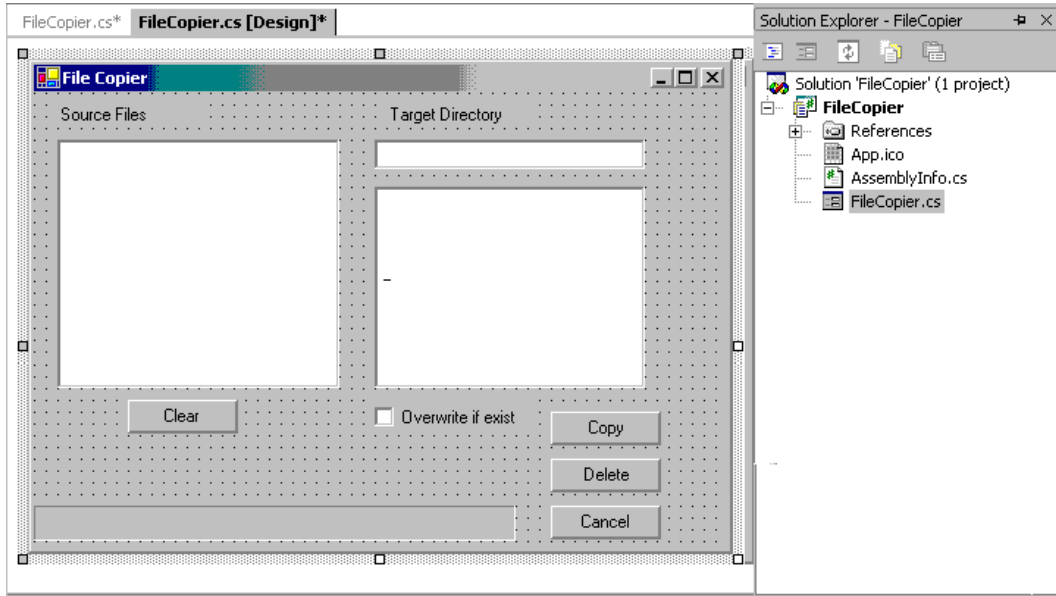


**Hình 2-28: Giao diện của FileCopier**

Mục tiêu của ứng dụng là cho phép người sử dụng đánh dấu check các tập tin (hoặc trọn thư mục) trên cây thư mục bên trái (tvwSource). Nếu người sử dụng ấn nút <Copy> thì các tập tin bị đánh dấu ở bên trái sẽ được chép qua Target Directory bên phải. Nếu người sử dụng ấn phím <Delete> thì các tập tin bị checked sẽ bị gỡ bỏ.

### 2.10.1 Tạo biểu mẫu giao diện cơ bản

Việc đầu tiên là ra lệnh **File | New | Project** để cho hiện lên khung đối thoại **New Project**, rồi chọn **Visual C# Projects** và **Windows Application**, đặt tên dự án là **FileCopier** và khô tên thư mục đối với dự án. Lúc này bạn ở chế độ Design. Trên **Solution Explorer**, bạn cho đổi tên chuẩn **Form1.cs** thành **FileCopier.cs**. Thông qua Toolbox, bạn cho lỗi thả các label, checkbox, button và tree view control, và bố trí thể nào giống như hình B dưới đây. Cuối cùng, bạn sử dụng cửa sổ Properties, thay đổi các thuộc tính **Name** và **Text** cho phù hợp với hình 2-29.



Hình 2-29: Giao diện FileCopier vào lúc thiết kế

Bạn muốn có những ô checkbox cạnh các tên tập tin và thư mục trên cây `twvSource` mà thôi. Như vậy bạn cho thuộc tính **CheckBoxes** của **twvSource** về `true` và **twvTargetDir** về `false`, trên cửa sổ Properties.

Một khi đã làm xong, bạn double click lên nút <Cancel> để tạo hàm thụ lý tình huống Click đối với nút <Cancel> này. Khi bạn double click lên một ô control, thì Visual Studio .NET IDE sẽ tạo ra một hàm thụ lý tình huống đối với đối tượng này. Một tình huống đặc biệt là target event, và Visual Studio .NET cho mở hàm thụ lý tình huống Click này:

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

Bạn có thể đặt để nhiều tình huống khác nhau đối với **TreeView** control, bằng lập trình bằng cách click lên nút Events (nút mang icon “sấm chớp”) trên cửa sổ Properties. Từ đây bạn có thể tạo những hàm thụ lý tình huống chỉ bằng cách điền tên hàm mà thôi. Visual Studio .NET sẽ ghi nhận tên hàm thụ lý tình huống và mở code editor nơi nó sẽ tạo header và con nháy sẽ nhấp nháy trong thân một hàm hành sự rỗng. Bạn có thể khở vào đoạn mã đối với hàm thụ lý tình huống. Thí dụ, trong trường hợp tình huống Click đối với nút <Cancel> chúng tôi khở vào lệnh `Application.Exit()`, cho biết là khi nào nút <Cancel> bị click thì thoát khỏi ứng dụng.

Visual Studio .NET sẽ kết sinh đoạn mã lo dàn dựng biểu mẫu và khởi gán tất cả các ô control, nhưng IDE sẽ không điền các ô control `TreeView`. Việc này bạn phải lo lấy bằng tay.

## 2.10.2 Cho điền các `TreeView` Control

Hai **`TreeView`** control hoạt động giống nhau, ngoại trừ, cây tay trái **`tvwSource`** sẽ cho liệt kê các thư mục và tập tin, trong khi cây tay phải **`tvwTargetDir`** thì chỉ liệt kê các thư mục. Thuộc tính **`CheckBoxes`** trên **`tvwSource`** được cho về **`true`**, còn trên **`tvwTargetDir`** thì được cho về **`false`**. Ngoài ra, mặc dù **`tvwSource`** cho phép chọn nhiều mục cùng một lúc (multiselect), là trị mặc nhiên đối với ô **`TreeView`** control, bạn sẽ cho **`tvwTargetDir`** về single selection.

Bạn sẽ cho dồn đoạn mã chung đối với **`TreeView`** control vào một hàm hành sự được chia sẻ sử dụng mang tên `FillDirectoryTree`, và chuyển cho ô control thông qua một flag chỉ cho biết cách lấy các tập tin. Bạn triệu gọi hàm từ hàm constructor, một lần cho mỗi **`TreeView`** control:

```
// Cho điền các cây thư mục nguồn và đích
FillDirectoryTree(tvwSource, true);
FillDirectoryTree(tvwTargerDir, false);
```

Hàm `FillDirectoryTree()` nhận thông số **`tvw`** là một **`TreeView`** control, khi thì là Source khi thì là Target. Hàm mang dáng dấp như sau:

```
private void FillDirectoryTree(TreeView tvw, bool isSource)
{
    . . .
}
```

### 2.10.2.1 Các đối tượng `TreeNode`

**`TreeView`** control có một thuộc tính **`Nodes`**, và thuộc tính này trả về một đối tượng collection **`TreeNodeCollection`**. Đây là một collection các đối tượng **`TreeNode`**, mỗi đối tượng này tượng trưng cho một mắt gút (node) trên cây. Ta bắt đầu cho xoá trắng collection này:

```
tvw.Nodes.Clear(); // xoá sạch các mắt gút
```

Bạn sẵn sàng điền collection **`Nodes`** của **`TreeView`** control bằng cách rảo lặp qua các thư mục của tất cả các ổ đĩa. Trước tiên, bạn lấy tất cả các ổ đĩa logic (A:, B:, C:, v.v..) được cài đặt trên hệ thống, do đó bạn triệu gọi một hàm static của đối tượng **`Environment`**, mang tên `GetLogicalDrives()`. Lớp **`Environment`** cung cấp thông tin liên quan đến môi trường của sàn diễn hiện hành cũng như việc truy cập môi trường này. Bạn



có thể dùng đối tượng **Environment** để lấy tên máy tính, phiên bản OS, thư mục hệ thống, v.v.. từ máy tính bạn đang chạy chương trình.

```
string[] strDrives = Environment.GetLogicalDrives();
```

**GetLogicalDrives()** trả về một bản dãy các chuỗi, mỗi chuỗi tượng trưng cho một thư mục gốc của một trong những ổ đĩa logic. Bạn sẽ rảo xem bản dãy này, rồi thêm các nút gút lên cho **TreeView** control khi bạn rảo qua:

```
foreach(string rootDirectoryName in strDrives)
{
```

Bạn phải xử lý mỗi ổ đĩa trong vòng lặp **foreach**. Điều đầu tiên phải làm là xác định liệu xem ổ đĩa đã sẵn sàng chưa. Bạn phải đi lấy danh sách các thư mục chóp bu từ ổ đĩa bằng cách triệu gọi hàm **GetDirectories()** trên một đối tượng **DirectoryInfo** mà chúng tôi đã tạo đối với thư mục gốc:

```
DirectoryInfo dir = new DirectoryInfo(rootDirectoryName);
dir.GetDirectories();
```

Lớp **DirectoryInfo** cho trưng những hàm hành sự thể hiện dùng tạo, di chuyển, và liệt kê xuyên qua các thư mục, các tập tin thuộc thư mục cũng như các thư mục con. Lớp **DirectoryInfo** đã được đề cập đến ở chương 1. Đề nghị bạn xem lại mục 1.1.2.2 trên chương 1.

Hàm **GetDirectories()** trả về một danh sách các thư mục, nhưng bạn sẽ “vắt vào sọt rác”, vì chẳng qua bạn triệu gọi hàm này để phát ra một biệt lệ nếu ổ đĩa chưa sẵn sàng. Bạn cho bao lệnh triệu gọi bởi một khối **try**, và không làm gì cả trong khối **catch**. Kết quả là nếu xuất hiện một biệt lệ thì ổ đĩa bị nhảy qua.

Một khi bạn biết được ổ đĩa đã sẵn sàng, bạn tạo một **TreeNode** lo giữ thư mục gốc của ổ đĩa và thêm nút gút vào **TreeView** control:

```
TreeNode ndRoot = new TreeNode(rootDirectoryName);
tvw.Nodes.Add(ndRoot);
```

Bây giờ bạn muốn rảo xuyên qua các thư mục, do đó bạn triệu gọi một hàm mới mang tên **GetSubDirectoryNodes()**, trao qua nút gút gốc, tên của thư mục gốc, và một flag cho biết liệu xem bạn muốn các tập tin hay không:

```
if(isSource)
{
    GetSubDirectoryNodes(ndRoot, ndRoot.Text, true);
}
else
{
    GetSubDirectoryNodes(ndRoot, ndRoot.Text, false);
}
```

```
}
```

Bạn tự hỏi vì sao phải trao qua **ndRoot.Text** nếu bạn đã trao **ndRoot** rồi. Bạn kiên nhẫn một chút. Bạn sẽ thấy vì sao phải cần đến khi bạn rảo lui về **GetSubDirectoryNodes**.

### 2.10.2.2 Rảo đệ quy xuyên qua các thư mục con

Hàm **GetSubDirectoryNodes** bắt đầu một lần nữa triệu gọi hàm **GetDirectories()**, nhưng lần này cất giấu đi bản dãy kết quả của các đối tượng **DirectoryInfo**:

```
private void GetSubDirectoryNodes(TreeNode parentNode,
                                string fullName, bool getFileNames)
{
    DirectoryInfo dir = new DirectoryInfo(fullName);
    DirectoryInfo[] dirSubs = dir.GetDirectories();
```

Bạn để ý mắt gút được trao qua cho hàm mang tên **parentNode**. Cấp các mắt gút hiện được xem xét sẽ được xem như là mắt gút con-cái đối với **parentNode** được trao qua. Đây là cách bạn ánh xạ cấu trúc thư mục đối với đẳng cấp cây **TreeView** control.

Cho rảo qua mỗi thư mục con, và cho nhảy qua bất cứ thư mục con nào được đánh dấu **Hidden**.

```
foreach (DirectoryInfo dirSub in dirSubs)
{
    if((dirSub.Attributes & FileAttributes.Hidden) != 0)
    {
        continue;
    }
    . . .
}
```

**FileAttributes** là một enumeration gồm những trị **Archive**, **Compressed**, **Directory**, **Encrypted**, **Hidden**, **Normal**, **ReadOnly**, v.v..

Bạn tạo một **TreeNode** với tên thư mục và thêm nó vào collection **Nodes** của mắt gút được trao cho hàm (**parentNode**):

```
TreeNode subNode = new TreeNode(dirSub.Name);
parentNode.Nodes.Add(subNode);
```

Bây giờ bạn đệ quy lại hàm **GetSubDirectoryNodes**, trao qua mắt gút bạn vừa mới tạo, **subNode**, như là mắt gút cha-mẹ mới, full path như là tên trọn vẹn của mắt gút cha-mẹ, và flag:

```
GetSubDirectoryNodes(subNode, dirSub.FullName, getFileNames);
```

**Bạn để ý:** Khi triệu gọi hàm constructor **TreeNode**, bạn dùng thuộc tính **Name** của đối tượng **DirectoryInfo**, trong khi triệu gọi hàm **GetSubDirectoryNodes()** bạn lại dùng thuộc tính **FullName**. Nếu thư mục của bạn là **C:\WinNT\Media\Sounds**, thì thuộc tính **FullName** sẽ trả về full path, trong khi thuộc tính **Name** chỉ trả về **Sounds**. Bạn chỉ trao qua tên mất gút vì bạn chỉ muốn hiển thị tên này lên tree view. Còn bạn trao full name với path cho hàm **GetSubDirectoryNodes()** để hàm này có thể lục ra tất cả các thư mục con trên ổ đĩa. Đây là câu trả lời cho câu hỏi đi trước vì sao cần trao qua tên mất gút gốc lần đầu tiên bạn triệu gọi hàm này; những gì trao qua không phải tên mất gút mà là full path đối với thư mục tương trưng bởi mất gút.

### 2.10.2.3 Đi lấy các tập tin trên thư mục

Một khi bạn đã rào đệ quy qua các thư mục con, đã đến lúc đi lấy các tập tin đối với thư mục, nếu flag **getFileNames = true**. Muốn thế, bạn triệu gọi hàm **GetFiles()** của đối tượng **DirectoryInfo**. Hàm này trả về một bản dãy các đối tượng **FileInfo**:

```
if (getFileNames)
{ // Đi lấy bất cứ tập tin nào trên mất gút này
    FileInfo[] files = dir.GetFiles();
```

Lớp **FileInfo** đã được đề cập đến ở chương 1, cung cấp những hàm thể hiện dùng thao tác các tập tin.

Bây giờ bạn có thể rào qua collection **files** này, truy cập thuộc tính **Name** của đối tượng **FileInfo** và trao tên cho hàm constructor của một **TreeNode**, rồi sau đó thêm vào collection **Nodes** của mất gút cha-mẹ (như vậy tạo một mất gút con-cái). Lần này không đệ quy vì các tập tin không có thư mục con:

```
foreach (FileInfo file in files)
{    TreeNode fileNode = new TreeNode(file.Name);
    parentNode.Nodes.Add(fileNode);
}
```

Đây là tất cả những gì phải làm để điền hai **TreeView** control.

## 2.10.3 Thụ lý các tình huống của TreeView

Trong thí dụ này, bạn phải thụ lý một số tình huống. Trước tiên, có thể người sử dụng click lên các nút Cancel, Copy, Delete hoặc Clear. Thứ hai, người sử dụng có thể click một trong những checkbox trên **TreeView** control bên trái hoặc một trong những mất gút

trên **TreeView** control bên phải.

Bây giờ ta thử xét đến việc người sử dụng click lên **TreeView** control.

### 2.10.3.1 Click TreeView nguồn

Có hai đối tượng **TreeView** control, mỗi đối tượng có riêng hàm thụ lý tình huống. Ta thử xét đến đối tượng **TreeView** nguồn. Người sử dụng click lên các thư mục và tập tin mà họ muốn chép từ đây đi. Mỗi lần người sử dụng click một tập tin hoặc một thư mục, một số tình huống sẽ xảy ra. Tình huống mà bạn phải thụ lý là **AfterCheck**.

Muốn thế, bạn cho thi công một hàm custom thụ lý tình huống mang tên **tvwSource\_AfterCheck**. Bạn có thể dùng IDE để tạo một hàm rỗng:

```
tvwSource.AfterCheck += new  
    System.Windows.Forms.TreeViewEventHandler(tvwSource_AfterCheck);
```

Thi công của **AfterCheck()** ủy nhiệm công việc cho một hàm hành sự có thể đệ quy được là **SetCheck()**, bạn cũng có thể viết:

```
protected void tvwSource_AfterCheck(object sender,  
    System.Windows.Forms.TreeViewEventArgs e)  
{  
    SetCheck(e.Node, e.Node.Checked);  
}
```

Hàm thụ lý tình huống trao qua đối tượng sender và một đối tượng kiểu dữ liệu **TreeViewEventArgs**. Xem ra bạn có thể lấy mắt gút từ đối tượng **e**. Bạn cho triệu gọi hàm **SetCheck()** trao qua mắt gút và tình trạng liệu xem mắt gút có bị checked hay không.

Mỗi đối tượng node có một thuộc tính **Nodes**, đi lấy một **TreeNodeCollection** chứa tất cả các subnode. **SetCheck()** rảo đệ quy xuyên qua collection **Nodes** của mắt gút hiện hành, cho đặt để check mark khớp với mắt gút bị checked. Nói cách khác, khi bạn cho check một thư mục, tất cả các tập tin và thư mục của nó sẽ bị checked theo đệ quy lần xuống.

Đối với mỗi **TreeNode** trong collection **Nodes**, bạn kiểm tra liệu xem là một lá (leaf) hay không. Nếu mắt gút là một leaf nếu collection **Nodes** của riêng nó sẽ có một cái đếm bằng zero. Nếu thế, bạn cho đặt để thuộc tính **Checked** về cái gì đó được trao qua như là thông số, còn nếu không phải là lá, thì bạn cho đệ quy lại. Sau đây là 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);
        }
    }
}
```

Việc này cho lan lần check mark (hoặc xoá check mark) xuống toàn bộ cấu trúc. Theo cách này, người sử dụng có thể cho biết họ muốn chọn ra tất cả các tập tin trong tất cả các thư mục con chỉ cần click lên một thư mục duy nhất.

### 2.10.3.2 Click *TreeView* đích

Hàm thụ lý tình huống đối với **TreeView** đích thì hơi rắc rối một chút. Hàm thụ lý phải quan tâm là tình huống **AfterSelect** (bạn nhớ cho là **TreeView** đích không có checkbox). Lần này, bạn muốn lấy một thư mục được chọn ra và cho đặt trọn path của nó vào text box ở trên đỉnh biểu mẫu, txtTargetDir.

Muốn thế, bạn phải làm việc từ dưới lên xuyên qua các mắt gút, tìm ra tên của thư mục cha-mẹ và xây dựng full path:

```
private void tvwTargetDir_AfterSelect(object sender,
                                     System.Windows.Forms.TreeViewEventArgs e)
{
    string theFullPath = GetParentString(e.Node);
```

Ta sẽ xem sau trong chốc lát hàm **GetParentString()**. Một khi bạn đã có full path, bạn phải tia xuống backslash (nếu có) về cuối và sau đó bạn cho điền ô text box txtTargetDir.

```
if (theFullPath.EndsWith("\\"))
{
    theFullPath = theFullPath.Substring(0, theFullPath.Length-1);
}
txtTargetDir.Text = theFullPath;
```

Hàm **GetParentString()** nhận một mắt gút và trả về một chuỗi với full path. Muốn thế, nó đệ quy lần lên path thêm backslash sau bất cứ mắt gút nào không phải là một leaf:

```
private string GetParentString(TreeNode node)
{
    if (node.Parent == null)
    {
        return node.Text;
```

```

    }
    else
    {
        return GetParentString(node.Parent) + node.Text +
            (node.Nodes.Count == 0 ? "": "\\");
    }
}

```

Đệ quy ngừng khi không còn mắt gút cha-mẹ, nghĩa là khi bạn đệ quy mắt gút gốc.

## 2.10.4 Thụ lý tình huống Click nút <Clear>

Vì ta đã triển khai hàm hành sự **SetCheck()** trước kia, do đó việc thụ lý tình huống Click của nút <Clear> là chuyển “dễ ợt”:

```

private void btnClear_Click(object sender, System.EventArgs e)
{
    foreach (TreeNode node in tvwSource.Nodes)
    {
        SetCheck(node, false);
    }
}

```

Bạn chỉ cần triệu gọi hàm hành sự **SetCheck()** trên mắt gút gốc và yêu cầu chúng uncheck một cách đệ quy tất cả các mắt gút được chứa,

## 2.10.5 Thi công tình huống Click nút <Copy>

Bây giờ bạn có thể đánh dấu check các tập tin và chọn lấy thư mục đích, bạn đã sẵn sàng thụ lý tình huống Click đối với nút <Copy>. Điều đầu tiên là bạn phải có danh sách các tập tin đã được chọn ra, nghĩa là bạn phải có một bản dãy các đối tượng **FileInfo**, nhưng không biết là có bao nhiêu đối tượng trong danh sách. Đây đúng là công việc dành cho **ArrayList**. Bạn ủy quyền điền danh sách cho một hàm hành sự mang tên **GetFileList()**:

```

protected void btnCopy_Click(object sender, System.EventArgs e)
{
    ArrayList fileList = GetFileList();
}

```

Ta thử thiết kế hàm hành sự này trước khi trở về hàm thụ lý tình huống.

### 2.10.5.1 Đi lấy các tập tin được tuyển chọn

Bạn bắt đầu bằng cách thể hiện một đối tượng **ArrayList** mới dùng chứa những chuỗi tên các tập tin được tuyển ra:

```
private ArrayList GetFileList()
{
    ArrayList fileNames = new ArrayList();
```

Muốn lấy tên các tập tin được tuyển ra, bạn có thể rảo qua **TreeView** nguồn:

```
foreach (TreeNode theNode in tvwSource.Nodes)
{
    GetCheckedFiles(theNode, fileNames);
}
```

Muốn biết việc chạy ra sao, ta thử đi vào hàm hành sự **GetCheckedFiles()**. Hàm này khá đơn giản: nó xem xét mắt gút được trao qua. Nếu mắt gút này không có con-cái (**node.Nodes.Count == 0**), thì là một lá. Nếu lá này mang dấu check, thì bạn muốn lấy full path (bằng cách triệu gọi hàm **GetParentString()** đối với node) và thêm nó vào **ArrayList** được trao qua như là một thông số

```
private void GetCheckedFiles(TreeNode node, ArrayList fileNames)
{
    if (node.Nodes.Count == 0)
    {
        if (node.Checked)
        {
            string fullPath = GetParentString(node);
            fileNames.Add(fullPath);
        }
    }
```

Nếu mắt gút không phải là lá, bạn muốn rảo đệ quy xuống lần theo cây, tìm các mắt gút con cái:

```
else
{
    foreach (TreeNode n in node.Nodes)
    {
        GetCheckedFiles(n, fileNames);
    }
}
```

Ta trở về **ArrayList** được điền đầy với tất cả các tên tập tin. Ta trở lui về hàm **GetFileList()**. Bạn sẽ dùng **ArrayList** các tên tập tin này để tạo một **ArrayList** thứ hai, lần này để chứa các đối tượng **FileInfo** hiện thời:

```
ArrayList fileList = new ArrayList();
```

Một lần nữa, bạn để ý là bạn không bảo hàm constructor **ArrayList** loại đối tượng nào nó sẽ chứa. Đây là một trong nhiều lợi điểm của một hệ thống kiểu dữ liệu có gốc rễ; collection chỉ biết là có một vài loại gì đó thuộc **Object**; vì tất cả các đối tượng đều được dẫn xuất từ **Object**, danh sách có thể chứa dễ dàng các đối tượng **FileInfo** cũng như có thể chứa các đối tượng kiểu string.

Bây giờ bạn có thể rảo lặp qua các tên tập tin trên **ArrayList**, trích ra mỗi tên và thể hiện một đối tượng **FileInfo** tương ứng. Bạn có thể phát hiện liệu xem nó là một tập tin hay là một thư mục bằng cách triệu gọi thuộc tính **Exists**, thuộc tính này trả về false nếu là

đối tượng **File** hiện bạn tạo ra là một thư mục. Nếu là một **File**, bạn có thể thêm nó vào **ArrayList** mới:

```
foreach (string fileName in fileNames)
{
    FileInfo file = new FileInfo(fileName);
    if (file.Exists)
    {
        fileList.Add(file);
    }
}
```

### 2.10.5.2 Cho sắp xếp các tập tin được chọn ra

Các tập tin được chọn ra phải được sắp xếp. Do đó phải sắp xếp **ArrayList**. Bạn có thể triệu gọi hàm `Sort()`, nhưng làm sao biết sắp xếp các đối tượng **File**? Bạn nhớ cho là **ArrayList** không tài nào biết được nội dung của nó.

Để giải quyết vấn đề, bạn phải chui vào một giao diện **IComparer**. Ta sẽ tạo một lớp mang tên **FileComparer** cho thi công giao diện này và nó sẽ biết cách sắp xếp các đối tượng **FileInfo**:

```
public class FileComparer: IComparer
{
```

Lớp này chỉ có duy nhất một hàm hành sự **Compare()** nhận hai thông số object:

```
public int Compare (object f1, object f2)
{
```

Tiếp cận thông thường (từ nhỏ đến lớn) là trả về 1 nếu đối tượng đầu (f1) lớn hơn đối tượng thứ hai (f2), trả về -1 nếu ngược lại, và trả về 0 nếu cả hai đối tượng bằng nhau. Tuy nhiên, trong trường hợp này, bạn muốn danh sách được sắp xếp từ lớn xuống nhỏ, bạn đảo trị trả về.

Muốn trắc nghiệm chiều dài của đối tượng **FileInfo**, bạn phải cast các thông số **Object** về các đối tượng **FileInfo**:

```
FileInfo file1 = (FileInfo) f1;
FileInfo file2 = (FileInfo) f2;
if(file1.Length > file2.Length)
{
    return -1;
}
if(file1.Length < file2.Length)
{
    return 1;
}
return 0;
}
}
```



**Bạn để ý:** Trong một chương trình thực tế sản xuất, có thể bạn phải trải nghiệm kiểu dữ liệu của đối tượng và có thể phải cho biết lệ nếu đối tượng không thuộc kiểu bạn chờ đợi.

Ta trở về **GetFileList()**, bạn chuẩn bị thể hiện qui chiếu **IComparer** và trao nó qua cho hàm hành sự **Sort()** của **fileList**:

```
IComparer comparer = (IComparer) new FileComparer();
fileList.Sort(comparer);
```

Xong việc, bạn có thể trả về **fileList** cho hàm triệu gọi:

```
return fileList;
```

Hàm triệu gọi là **btnCopy\_Click**. Bạn còn nhớ, bạn nhảy về **GetFileList()** khi đang ở dòng lệnh đầu tiên của hàm thụ lý tình huống **btnCopy\_Click()**.

```
protected void btnCopy_Click(object sender, System.EventArgs e)
{
    ArrayList fileList = GetFileList();
```

Tới lúc này, bạn trở về với một danh sách các đối tượng **File** được sắp xếp, mỗi đối tượng tương trưng cho một tập tin được tuyển chọn trên **TreeView** nguồn. Bây giờ, bạn có thể rảo laps qua danh sách, chép các tập tin và nhật tu giao diện:

```
foreach (FileInfo file in fileList)
{
    try
    {
        lblStatus.Text = "Copying " + txtTargetDir.Text +
            "\\\" + file.Name + "...";
        Application.DoEvents();
        file.CopyTo(txtTargetDir.Text + "\\\" + file.Name,
            chkOverwrite.Checked);
    }
    catch (Exception ex)
    {
        MessageBox.Show(ex.Message);
    }
}
lblStatus.Text = "Done.";
Application.DoEvents();
```

Khi bạn chạy chương trình, tiến triển công việc được viết lên một label, mang tên **lblStatus**. Việc triệu gọi hàm **Application.DoEvents()** cho phép UI một cơ may tô vẽ lại. Sau đó, bạn triệu gọi hàm **CopyTo()** đối với một tập tin, trao qua thư mục đích, được nhận từ ô text box, và một flag bool cho biết tập tin phải bị viết chồng lên hay không nếu nó đã hiện hữu.

Bạn để ý là flag bạn trao qua là trị của ô duyệt **chkOverwrite**. Thuộc tính **Checked** sẽ định trị true nếu ô check box có dấu check, và false nếu không có dấu check.

Phần sao chép được đặt nằm trong khối try vì bạn có thể tiên liệu nhiều điều trắc trở sẽ xảy ra khi sao chép các tập tin. Hiện giờ, thì bạn thụ lý các biệt lệ bằng cách cho hiện lên một khung đối thoại với sai lầm, nhưng trong các ứng dụng mang tính thương mại, bạn phải trù liệu những bước sửa sai.

Coi như xong việc sao chép các tập tin với nút <Copy>.

## 2.10.6 Thi công tình huống Click nút <Delete>

Hàm thụ lý tình huống Click trên nút <Delete> không có chi rắc rối. Điều đầu tiên là hỏi người sử dụng có chắc hủy bỏ các tập tin hay không:

```
protected void btnDelete_Click(object sender, System.EventArgs e)
{
    System.Windows.Forms.DialogResult result =
        MessageBox.Show("Are you quite sure?", // msg
            "Delete Files", // caption
            MessageBoxButtons.OKCancel, // buttons
            MessageBoxIcon.Exclamation, // icon
            MessageBoxDefaultButton.Button2); // default button
```

Khi người sử dụng chọn OK hoặc Cancel, kết quả chuyển trả lại như là một trị enum của DialogResult. Bạn có thể trắc nghiệm trị này để xem người sử dụng có ấn phím OK hay không:

```
if (result == System.Windows.Forms.DialogResult.OK)
{
```

Nếu là OK thì bạn có thể lấy danh sách của fileNames và rảo laps qua danh sách này và cho gỡ bỏ mỗi tập tin bạn tìm thấy:

```
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 toàn bộ dự án FileCopier. Chúng tôi bỏ qua những phần kết sinh của IDE về giao diện:

```
using System;  
using System.Drawing;  
using System.Collections;  
using System.ComponentModel;  
using System.Windows.Forms;  
using System.IO;  
using System.Data;  
  
namespace FileCopier  
{  
    public class Form1: System.Windows.Forms.Form  
    {  
        private System.Windows.Forms.TreeView tvwSource;  
        private System.Windows.Forms.TreeView tvwTargetDir;  
        private System.Windows.Forms.Button btnClear;  
        private System.Windows.Forms.Button btnCopy;  
        private System.Windows.Forms.Button btnDelete;  
        private System.Windows.Forms.Button btnCancel;  
        private System.Windows.Forms.Label lblSource;  
        private System.Windows.Forms.Label lblTargetDir;  
        private System.Windows.Forms.TextBox txtTargetDir;  
        private System.Windows.Forms.CheckBox chkOverwrite;  
        private System.Windows.Forms.Label lblStatus;  
  
        // So sánh hai tập tin từ lớn xuống nhỏ  
        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();  
            // Cho điền các cây thu mục nguồn và đích  
            FillDirectoryTree(tvwSource, true);  
            FillDirectoryTree(tvwTargetDir, false);  
        }  
    }  
}
```

```

// Điền cây thư mục Source hoặc Target
private void FillDirectoryTree(TreeView tvw, bool isSource)
{
    tvw.Nodes.Clear(); // xoá sạch các mắt gút
    // Điền bản dãy với tên ổ đĩa logic của máy tính
    string[] strDrives = Environment.GetLogicalDrives();
    // Rào lặp qua các ổ đĩa, thêm chúng vào treeview
    foreach(string rootDirectoryName in strDrives)
    { if (rootDirectoryName != @"D:\")
        continue;
        try
        { // Điền bản dãy với các thư mục con cấp 1
            DirectoryInfo dir = new
                DirectoryInfo(rootDirectoryName);
            dir.GetDirectories();
            TreeNode ndRoot = new TreeNode(rootDirectoryName);
            tvw.Nodes.Add(ndRoot); // thêm một node cho mỗi root
                                // directory
            // Thêm subdirectory node. Nếu treeview là nguồn thì
            // lấy thêm tên các tập tin
            if(isSource)
            { GetSubDirectoryNodes(ndRoot, ndRoot.Text, true);
            }
            else
            { GetSubDirectoryNodes(ndRoot, ndRoot.Text, false);
            }
        }
        catch (Exception e)
        { MessageBox.Show(e.Message);
        }
    }
} // end FillSourceDirectoryTree

// Đi lấy tất cả các thư mục con
private void GetSubDirectoryNodes(TreeNode parentNode,
    string fullName, bool getFileNames)
{
    DirectoryInfo dir = new DirectoryInfo(fullName);
    DirectoryInfo[] dirSubs = dir.GetDirectories();

    // Thêm một child node đối với mỗi subdirectory
    foreach (DirectoryInfo dirSub in dirSubs)
    { if((dirSub.Attributes & FileAttributes.Hidden) != 0)
        { continue;
        }

        TreeNode subNode = new TreeNode(dirSub.Name);
        parentNode.Nodes.Add(subNode);
        GetSubDirectoryNodes(subNode, dirSub.FullName,
            getFileNames);
    }
    if (getFileNames)
    { FileInfo[] files = dir.GetFiles();
        foreach (FileInfo file in files)

```

```
        {   TreeNode fileNode = new TreeNode(file.Name);
            parentNode.Nodes.Add(fileNode);
        }
    } // end GetSubDirectoryNodes

private void InitializeComponent()
{
    // (mã nguồn kết sinh bởi IDE đối với giao diện ở đây)
    ...
}

private void btnClear_Click(object sender, System.EventArgs e)
{
    foreach (TreeNode node in tvwSource.Nodes)
    {   SetCheck(node, false);
    }
}

protected void btnCopy_Click(object sender, System.EventArgs e)
{
    ArrayList fileList = GetFileList();
    foreach (FileInfo file in fileList)
    {   try
        {   lblStatus.Text = "Copying " + txtTargetDir.Text + "\\\" +
            file.Name + "...";
            Application.DoEvents();
            file.CopyTo(txtTargetDir.Text + "\\\" + file.Name,
                chkOverwrite.Checked);
        }
        catch (Exception ex)
        {   MessageBox.Show(ex.Message);
        }
    }
    lblStatus.Text = "Done.";
    Application.DoEvents();
}

private ArrayList GetFileList()
{
    ArrayList fileNamees = new ArrayList();
    foreach (TreeNode theNode in tvwSource.Nodes)
    {   GetCheckedFiles(theNode, fileNamees);
    }

    ArrayList fileList = new ArrayList();
    foreach (string fileName in fileNamees)
    {   FileInfo file = new FileInfo(fileName);
        if (file.Exists)
        {   fileList.Add(file);
        }
    }

    IComparer comparer = (IComparer) new FileComparer();
    fileList.Sort(comparer);
    return fileList;
}
```

```

    }

    private void GetCheckedFiles(TreeNode node, ArrayList fileNames)
    {
        if (node.Nodes.Count == 0)
        {
            if (node.Checked)
            {
                string fullPath = GetParentString(node);
                fileNames.Add(fullPath);
            }
            else
            {
                foreach (TreeNode n in node.Nodes)
                {
                    GetCheckedFiles(n, fileNames);
                }
            }
        }
    }

    private void btnDelete_Click(object sender, System.EventArgs e)
    {
        System.Windows.Forms.DialogResult result = MessageBox.Show(
            "Are you quite sure?",
            "Delete Files",
            MessageBoxButtons.OKCancel,
            MessageBoxIcon.Exclamation,
            MessageBoxDefaultButton.Button2);
        if (result == System.Windows.Forms.DialogResult.OK)
        {
            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();
        }
    }

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

    private void tvwSource_AfterCheck(object sender,
        System.Windows.Forms.TreeViewEventArgs e)
    {
        SetCheck(e.Node, e.Node.Checked);
    }

    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);
            }
        }
    }
}

private void tvwTargetDir_AfterSelect(object sender,
    System.Windows.Forms.TreeViewEventArgs e)
{
    string theFullPath = GetParentString(e.Node);
    if (theFullPath.EndsWith("\\\\"))
    {
        theFullPath = theFullPath.Substring(0,
            theFullPath.Length - 1);
    }
    txtTargetDir.Text = theFullPath;
}

private string GetParentString(TreeNode node)
{
    {
        if (node.Parent == null)
        {
            return node.Text;
        }
        else
        {
            return GetParentString(node.Parent) + node.Text +
                (node.Nodes.Count == 0 ? "": "\\");
        }
    }
}
}

```

## 2.11 Tương tác với System Registry

Nếu bạn là dân lập trình COM, thì bạn không thể tránh (sự khốn khổ đối với) Registry của Windows. Còn nếu bạn sống chung với Visual Studio .NET thì việc đi lại với **Registry** rất ư thừa thớt. NET Framework đã giảm sự quan trọng của **Registry** đối với ứng dụng, vì assembly đã trở thành “tự cung tự cấp” do đó assembly không cần thông tin đặc biệt được trữ trên **Registry**. **Registry** giờ đây chỉ là nơi tiện lợi để bạn trữ thông tin về sở thích của người sử dụng (user preference).

Namespace **Microsoft.Win32** định nghĩa một vài lớp cho phép đọc (hoặc viết) system registry một cách dễ dàng. Bảng 2-25 liệt kê các lớp này:

**Bảng 2-25: Các lớp thao tác System Registry**

Lớp Microsoft.Win32	Ý nghĩa
<b>Registry</b>	Một trừu tượng hóa cấp cao về bản thân <b>Registry</b> , với tất cả các lớp được gắn liền.
<b>RegistryKey</b>	Đây là lớp cốt lõi, cho phép bạn thêm vào, gỡ bỏ và nhật tu thông tin được trữ trong <b>Registry</b> .
<b>RegistryHive</b>	Đây chỉ là liệt kê mỗi khuôn tổ ong (hive) trên <b>Registry</b>

Trước khi ta xem xét các lớp kể trên, chúng tôi rào qua xem bản thân cấu trúc của **Registry**.

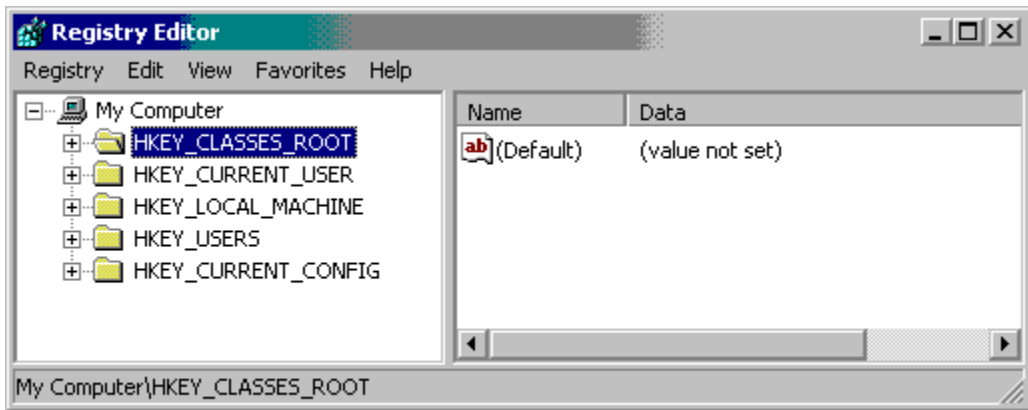
## 2.11.1 Registry

Registry có một cấu trúc đẳng cấp giống như hệ thống các tập tin (file system). Cách thông thường để nhìn xem hoặc thay đổi nội dung của Registry là với một trong hai trình tiện ích: **regedit.exe** hoặc **regedt32.exe**. **regedit.exe** hiện diện trong tất cả các phiên bản Windows, từ khi Windows 95 trở thành chuẩn. Còn **Regedt32.exe** thì chỉ hiện diện trong Windows NT và Windows 2000, ít thân thiện so với **regedit.exe** nhưng cho phép truy cập vào thông tin an ninh mà **regedit** không có khả năng nhìn xem. Trong phần này, chúng tôi sử dụng **regedit.exe** tại khung đối thoại **Run** hoặc command prompt.

Khi bạn khởi động **regedit.exe** từ khung đối thoại **Start | Run...** thì bạn có khung đối thoại như sau (hình 2-30). Registry có giao diện mang dáng dấp treeview/listview giống như với Windows Explorer, khớp với cấu trúc đẳng cấp của bản thân Registry. Tuy nhiên, như chúng ta sẽ thấy, có một vài khác biệt.

Trong một file system, các mắt gút cấp chóp bu có thể được xem như là những partitions trên ổ đĩa C:\, D:\ v.v.. Trong Registry, tương đương với partition là **registry hive** (khuôn tổ ong trên registry). Các khuôn này là cố định không thể thay đổi và có cả thảy là 7 (mặc dù chỉ thấy 5 trên **regedit.exe**).





**Hình 2-30: Khung đối thoại Registry Editor**

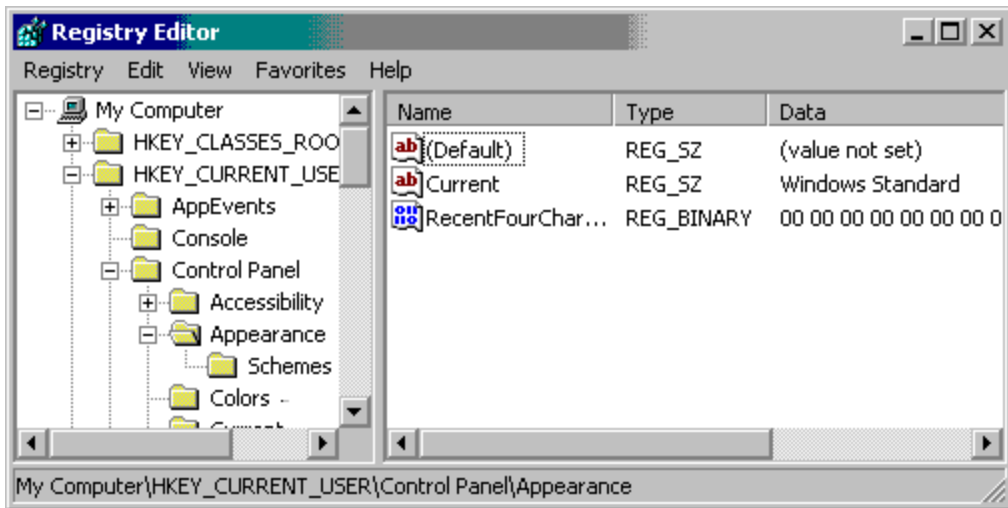
- HKEY\_CLASSES\_ROOT (HKCR): chứa những chi tiết về các loại tập tin (.txt, .doc, v.v..) và những ứng dụng nào có khả năng mở các tập tin loại nào. Ngoài ra, nó còn chứa thông tin đăng ký đối với tất cả các cấu kiện COM (chiếm phần lớn Registry, vì Windows mang theo vô số cấu kiện COM).
- HKEY\_CURRENT\_USER (HKCU) chứa những chi tiết liên quan đến những sở thích của người sử dụng hiện được đăng nhập trên máy tính.
- HKEY\_LOCAL\_MACHINE (HKLM) là một hive đồ sộ chứa những chi tiết của tất cả các phần mềm và phần cứng được cài đặt trên máy tính. Nó cũng bao gồm HKCR hive: HKCR hiện thật sự không phải là một hive độc lập tự thân, nhưng đơn giản là một ánh xạ tiện lợi trên registry key, HKLM/SOFTWARE/Classes.
- HKEY\_USERS (HKUSR) chứa những chi tiết liên quan đến sở thích của tất cả các người sử dụng. Như bạn có thể chờ đợi, nó cũng chứa hive HKCU, đơn giản là một ánh xạ lên một trong những key trên HKEY\_USERS.
- HKEY\_CURRENT\_CONFIG (HKCF) chứa những chi tiết liên quan đến phần cứng trên máy tính.

Phần còn lại là hai key chứa thông tin mang tính tạm thời và thay đổi thường xuyên:

- HKEY\_DYN\_DATA là một container tổng quát đối với bất cứ dữ liệu volatile nào cần trữ đầu đó trên Registry.
- HKEY\_PERFORMANCE\_DATA chứa thông tin liên quan đến thành tích của ứng dụng đang chạy.

Trong lòng các hive là một cấu trúc cây gồm các registry key. Mỗi key (mục khoá) cũng giống như một folder hoặc file trong một file system. Tuy nhiên, có một khác biệt rất quan trọng: file system phân biệt giữa các file (chứa dữ liệu) và folder (chủ yếu dùng chứa các file hoặc folder khác), nhưng Registry hiện diện chỉ toàn là key. Một key có thể chứa cả dữ liệu lẫn các key khác.

Nếu một key chứa dữ liệu, thì lúc này nó sẽ hiện diện như là một loạt các trị. Mỗi trị sẽ có một cái tên, một kiểu dữ liệu và một trị. Ngoài ra, một key có thể có một trị mặc nhiên không được đặt tên.



Hình 2-31: Key HKCU\ControlPanel\Appearance

Ta có thể thấy cấu trúc này bằng cách sử dụng **regedit.exe** để quan sát registry key. Hình 2-31 cho thấy nội dung của key HKCU\ControlPanel\Appearance, chứa chi tiết của color scheme được chọn của người sử dụng hiện được đăng nhập. regedit.exe cho thấy key nào được xem xét bằng cách cho nó hiển thị với một folder icon được mở trong tree view.

Key HKCU\Control Panel\Appearance có 3 bộ trị có mang tên, mặc dù trị mặc nhiên không chứa bất cứ dữ liệu nào. Cột Type, trên hình 2-31 chi tiết hoá kiểu dữ liệu của mỗi trị. Các mục vào Registry có thể được định dạng theo một trong 3 kiểu dữ liệu: REG\_SZ (gần như tương với .NET string), REG\_DWORD (gần như tương đương với .NET uint) và REG\_BINARY (bản dãy các byte).

## 2.11.2 Các lớp .NET Registry

Việc truy cập vào Registry trên .NET sẽ thông qua hai lớp **Registry** và **RegistryKey** thuộc namespace **Microsoft.Win32** (bảng 2-25). Một thể hiện của lớp **RegistryKey** tượng trưng cho một registry key. Lớp **RegistryKey** cung cấp những thành viên cốt lõi

sau đây (bảng 2-26) cho phép bạn làm việc với một registry key,

**Bảng 2-26: Các thành viên cốt lõi của lớp RegistryKey**

Các thành viên	Ý nghĩa
<b>Name</b>	Thuộc tính này tìm lại tên của key (read-only).
<b>SubKeyCount</b>	Thuộc tính này tìm lại cái đếm (count) số lượng subkey.
<b>ValueCount</b>	Thuộc tính này tìm lại cái đếm các trị trên một key.
<b>Close()</b>	Hàm này cho đóng lại key này và tuôn ghi nó lại lên đĩa nếu nội dung bị thay đổi.
<b>CreateSubKey()</b>	Hàm này tạo một subkey mới hoặc mở một subkey hiện hữu. Chuỗi subKey không phân biệt chữ hoa chữ thường.
<b>DeleteSubKey()</b>	Hàm này cho gỡ bỏ một subkey được chỉ định. Nếu muốn gỡ bỏ những subkey con-cái, thì sử dụng đến <b>DeleteSubKeyTree()</b> . Chuỗi subKey không phân biệt chữ hoa chữ thường.
<b>DeleteSubKeyTree()</b>	Hàm này cho gỡ bỏ một cách đệ quy một subkey và bất cứ subkey con-cái nào. Chuỗi subKey không phân biệt chữ hoa chữ thường.
<b>GetSubKeyNames()</b>	Hàm này cho tìm lại một bản dãy chuỗi chứa tất cả các tên của subkey.
<b>GetValue()</b>	Overloaded. Hàm này tìm lại trị được khai báo.
<b>GetValueNames()</b>	Hàm này cho tìm lại một bản dãy chuỗi chữ chứa tất cả các tên trị (value name).
<b>OpenRemoteBaseKey()</b>	Hàm này cho mở một RegistryKey mới tượng trưng cho key được yêu cầu trên một máy nằm ngoài.
<b>OpenSubKey()</b>	Overloaded. Hàm này cho tìm lại subkey.
<b>SetValue()</b>	Hàm này cho đặt để một trị được chỉ định. Chuỗi subKey không phân biệt chữ hoa chữ thường.

Lớp **RegistryKey** sẽ là lớp mà bạn sẽ dùng để làm việc với registry key. Ngược lại, lớp **Registry** là lớp mà bạn chỉ bao giờ thể hiện. Vai trò của nó là cung cấp cho bạn những thể hiện **RegistryKey** tượng trưng cho key chóp bu - những hive khác nhau - để có thể bắt đầu leo lái xuyên qua Registry. **Registry** cung cấp những thể hiện này thông qua các thuộc tính static, và có cả bảy thuộc tính bao gồm **ClassRoot**, **CurrentConfig**, **CurrentUser**, **DynData**, **LocalMachine**, **PerformanceData**, và **Users**. Chắc bạn đã biết thuộc tính nào chỉ registry hive nào. Do đó, muốn có một thể hiện của một **RegistryKey** tượng trưng cho key HKLM, bạn viết:

```
RegistryKey Hkml = Registry.LocalMachine;
```

Nếu bạn muốn đọc một vài dữ liệu trên key HKLM\Software\Microsoft, bạn phải đi lấy qui chiếu về key như sau:

```
RegistryKey Hkml = Registry.LocalMachine;  
RegistryKey HkSoftware = Hkml.OpenSubKey("Software");  
RegistryKey HkMicrosoft = HkSoftware.OpenSubKey("Microsoft");
```

Một **Registry key** được truy cập theo kiểu này chỉ cho phép bạn đọc mà thôi. Nếu bạn muốn có khả năng viết lên key (bao gồm viết lên trị của key, hoặc tạo hoặc gỡ bỏ key con-cái thuộc quyền), bạn phải sử dụng một **OpenSubkey** bị phủ quyết, nhận thêm một thông số thứ hai kiểu bool cho biết quyền read-write đối với key. Thí dụ, bạn muốn có khả năng thay đổi key **Microsoft**:

```
RegistryKey Hkml = Registry.LocalMachine;  
RegistryKey HkSoftware = Hkml.OpenSubKey("Software");  
RegistryKey HkMicrosoft = HkSoftware.OpenSubKey("Microsoft", true);
```

Hàm hành sự **OpenSubKey()** là một trong những hàm bạn sẽ triệu gọi nếu bạn chờ đợi key hiện hữu. Nếu nó không có, thì nó sẽ trả về một null reference. Còn nếu bạn muốn tạo một key mới, bạn sẽ dùng **CreateSubKey()** (hàm này tự động cho quyền read-write):

```
RegistryKey Hkml = Registry.LocalMachine;  
RegistryKey HkSoftware = Hkml.OpenSubKey("Software");  
RegistryKey HkMine = HkSoftware.CreateSubKey("MyOwnSoftware");
```

Điều thông thường thường xảy ra là ứng dụng của bạn cần bảo đảm là một vài dữ liệu hiện diện trong Registry - nói cách khác là tạo những key mang ý nghĩa nếu chúng chưa hiện diện, nhưng không làm gì cả nếu chúng hiện diện. **CreateSubkey()** đáp ứng đúng nhu cầu này.

Một khi bạn đã có **Registry key** bạn muốn đọc hoặc thay đổi, bạn có thể sử dụng các hàm hành sự **SetValue()** hoặc **GetValue()** để đặt dữ liệu hoặc đi lấy dữ liệu trên key. Thí dụ:

```
RegistryKey HkMine = HkSoftware.CreateSubKey("MyOwnSoftware");  
HkMine.SetValue("MyStringValue", "Hello World");
```

```
HkMine.SetValue("MyIntValue", 20);
```

Đoạn mã trên sẽ đặt để key về hai trị: MyStringValue sẽ mang kiểu dữ liệu REG\_SZ, trong khi MyIntValue sẽ mang kiểu dữ liệu REG\_DWORD.

RegistryKey.GetValue() hoạt động cũng như vậy. Nó được định nghĩa trả về một qui chiếu đối tượng, nghĩa là trả về một qui chiếu string nếu nó thấy có kiểu dữ liệu REG\_SZ và một int nếu phát hiện kiểu dữ liệu REG\_DWORD:

```
string StringValue = (string)HkMine.GetValue("MyStringValue");
int IntValue = (int)HkMine.GetValue("MyIntValue");
```

Cuối cùng, khi bạn xong việc, thì phải cho đóng lại:

```
HkMine.Close();
```

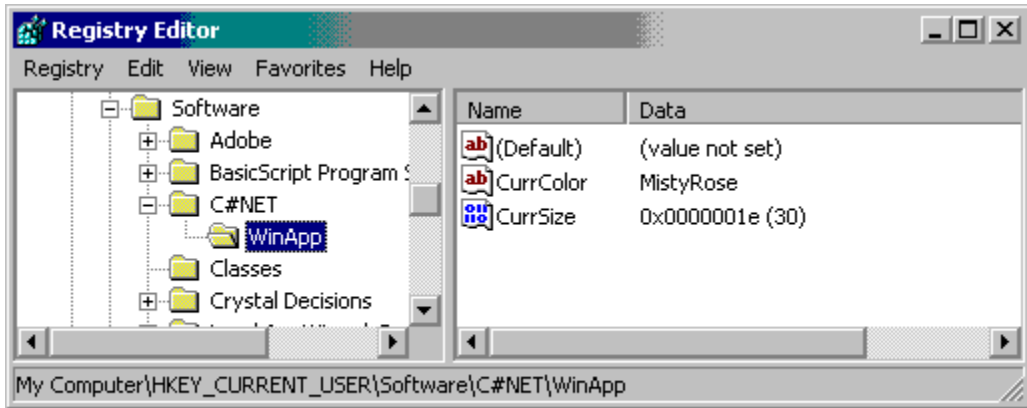
Bây giờ, mục tiêu của ứng dụng hiện hành là cho phép người sử dụng cất trữ những sở thích (preference) của họ (chẳng hạn phong chữ, kích thước phong chữ, v.v..) vào Registry để dùng về sau. Muốn thế, bạn phải sử dụng đến lớp **RegistryKey**,

Giả sử bạn có thêm một mục chọn trình đơn mới **"File | Save"**. Khi mục chọn này được chọn, bạn sẽ tạo một đối tượng **RegistryKey** và cho chèn màu background hiện hành và kích thước phong chữ vào HKEY\_CURRENT\_USER\Software\C#NET\WinApp. Ngoài ra, ta cũng giả định biểu mẫu của bạn có hai biến thành viên (currFontSize và currColor) trữ thông tin liên quan đến kích thước phong chữ hiện hành cũng như màu nền. Sau đây là đoạn mã, với cách sử dụng đến hàm **RegistryKey.SetValue()**:

```
// Giả định bạn có dữ liệu tình trạng như sau:
// Color currColor = Color.MistyRose;
// private int currFontSize = TheFontSize.Normal;

private void FileSave_Clicked(object sender, EventArgs e)
{ // cho cất trữ những thông tin sở thích lên Registry
  RegistryKey regKey = Registry.CurrentUser;
  regKey = regKey.CreateSubKey("Software\\C#NET\\WinApp");
  regKey.SetValue("CurrSize", currFontSize);
  regKey.SetValue("CurrColor", currColor.Name);
}
```

Nếu bây giờ người sử dụng cho màu hiện hành về màu LemonChiffon và kích thước phong chữ về 30 (và cho trữ những đặt để), bạn sẽ thấy những thông tin sau đây trên system registry như theo hình 2-32



**Hình 2-32: Cất trữ dữ liệu ứng dụng vào HKCU**

Đọc thông tin này từ registry cũng phải sử dụng đến lớp **RegistryKey**. Ta thử viết lại hàm constructor của biểu mẫu để đọc màu background và kích thước phông chữ từ registry rồi gán những biến thành viên tương ứng về đúng những trị. Theo cách này, ứng dụng khởi động với dáng dấp như theo châu làm việc đi trước. Bạn ghi nhận việc sử dụng đến hàm **RegistryKey.GetValue()**:

```
public MainForm()
{
    // Mở một subkey
    RegistryKey regKey = Registry.CurrentUser;
    regKey = regKey.CreateSubKey("Software\\C#NET\\WinApp");

    // Đọc vào dữ liệu rồi gán cho biến tình trạng
    currFontSize = (int)regKey.GetValue("CurrSize", currFontSize);
    string c = (string)regKey.GetValue("CurrColor", currColor.Name);
    currColor = Color.FromName(c);
    BackColor = currColor;

    ...
}
```

Một câu hỏi có thể được đặt ra: nếu hiện hành không có một mục vào đối với những data points trên registry, thì sao đây? Thí dụ, giả định người sử dụng khởi động lần đầu tiên ứng dụng và chưa đặt để gì cả. Trong trường hợp này, khi gặp phải hàm constructor, đối tượng **RegistryKey** chưa có thể tìm đúng dữ liệu.

Điều tốt lành là hàm **GetValue()** có thể nhận một thông số thứ hai tùy chọn. Thông số này cho biết trị được dùng thay thế khi gặp phải mục vào trống rỗng. Bạn để ý là bạn đã đưa dữ liệu vào các biến thành viên currFontSize và currColor. Nếu biểu mẫu phải cho

đặt để những biến này về những trị ban đầu, thì những biến này sẽ được sử dụng đến thay vì bất cứ trị mục vào registry thiếu vắng:

```
public class MainForm: Form
{
    Color currColor = Color.MistyRose;
    private int currFontSize = TheFontSize.Normal
    ...
}
```

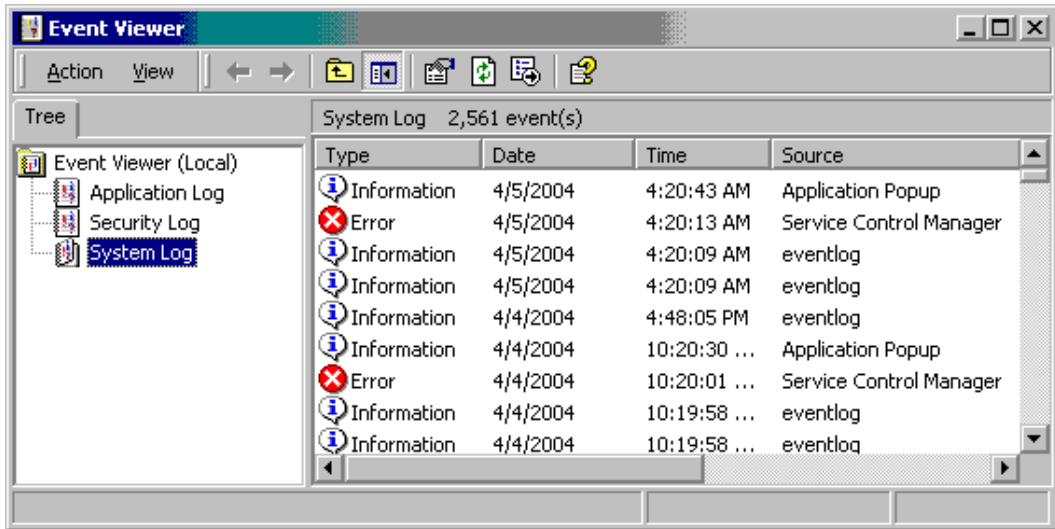
Việc làm cuối cùng là cho nhật tu hàm hỗ trợ **BuildMenuSystem()** kiểm tra đúng subitem trên trình đơn popup dựa trên thông tin được đọc vào từ registry. Trong ứng dụng trình đơn popup đi trước, bạn khai báo mục chọn trình đơn hiện hành được tuyển là TheFontSize.Normal. Điều này có thể là không thể nữa, nếu ta để ý đến việc người sử dụng có thể đã cho cất trữ sở thích của mình vào registry. Sau đây là đoạn mã cần nhật tu

```
private void BuildMenuSystem()
{
    ...
    // kích thước hiện hành ?
    if(currFontSize == TheFontSize.Huge)
        currentCheckedItem = checkedHuge;
    else if(currFontSize == TheFontSize.Normal)
        currentCheckedItem = checkedNormal;
    else
        currentCheckedItem = checkedTiny;
    currentCheckedItem.Checked = true;
}
```

## 2.12 Tương tác với Event Viewer

Windows 2000 cung cấp một MMC (Microsoft Management Console) mang tên “Event Viewer”. Event Viewer duy trì 3 log (bộ phận theo dõi) riêng rẽ: **Application log**, **Security log** và **System log**. Đây là cách bạn có thể lấy thông tin liên quan đến hardware, software và các vấn đề hệ thống và điều khiển những tình huống khác nhau về security. như theo hình 2-33.

Muốn khởi động **Event Viewer** bạn cho hiện lên cửa sổ **Server Explorer** rồi tìm ra mắt gút **Event Logs** bằng cách bung mắt gút **Servers**. Bạn cho right click lên **Event Logs** rồi chọn mục **Launch Event Viewer** thì cửa sổ **Event Viewer** hiện lên như theo hình 2-33



Hình 2-33: Win2000 Event Viewer

Khi bạn muốn lập trình để thao tác **Event Viewer**, bạn sẽ sử dụng đến những lớp khác nhau được định nghĩa trong namespace **System.Diagnostics**. Bảng 2-27 cho thấy những lớp cốt lõi của namespace này:

Bảng 2-27: Các lớp của namespace *System.Diagnostics*

Lớp của System.Diagnostics	Ý nghĩa
<b>EventLog</b>	Lớp này giúp bạn đột nhập vào Windows 2000 Event Viewer để thao tác.
<b>EventLog.EventLogEntryCollection</b>	Lớp này cầm giữ những lớp riêng rẽ EventLogEntry tượng trưng cho một mục vào (entry) trên một event log nào đó.
<b>EventLogEntry</b>	Lớp này tượng trưng cho một mẫu tin đơn lẻ (single record) trên event log.
<b>EventLogNames</b>	Lớp sealed này cung cấp những vùng mục tin (field) định nghĩa log mà bạn muốn thao tác (Application, Security hoặc System).

Sử dụng đến **EventLog**, bạn có thể đọc vào từ những log hiện hữu (Application, Security hoặc System), viết những mục vào lên log, gỡ bỏ log, và phản ứng trước những mục vào mà log nhận được. Nếu muốn, bạn có thể tạo một custom log mới khi tạo một event source (nguồn lực tình huống). Bảng 2-28 liệt kê một số thành viên cốt lõi của lớp **EventLog**.



**Bảng 2-28: Các thành viên cốt lõi của lớp EventLog**

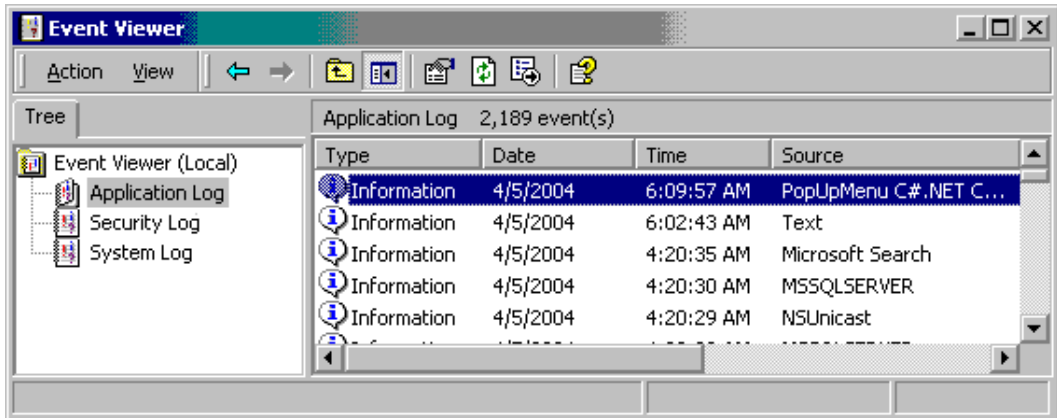
Thành viên	Ý nghĩa
<b>Entries</b>	Đi lấy nội dung của event log, cho giữ trong một lớp <b>EventLog.EventLogEntryCollection</b> . Lớp collection này chứa những mục <b>EventLogEntry</b> riêng biệt.
<b>Log</b>	Đi lấy hoặc đặt để tên log. Đây có thể là “Application”, “System” hoặc “Security” hoặc tên một custom log
<b>MachineName</b>	Đi lấy hoặc đặt để tên máy tính mà ta sẽ đọc hoặc viết event log. Nếu bạn không khai báo thì máy tại chỗ (“.”) sẽ được chọn.
<b>Source</b>	Đi lấy hoặc đặt để tên ứng dụng (source name) để đăng ký hoặc sử dụng đến khi viết event log.
<b>Clear()</b>	Cho gỡ bỏ tất cả các mục vào khỏi event log.
<b>Close()</b>	Cho đóng lại một log và giải phóng các mục quản (handle) đọc và viết.
<b>CreateEventSource()</b>	Thiết lập một ứng dụng như là một nguồn lực event.
<b>GetEventLogs()</b>	Tạo một bản dãy chứa các event log.
<b>WriteEntry(0</b>	Cho chèn một mục vào lên event log.

Như đã nói, ứng dụng của bạn sẽ viết một entry vào Application log khi ứng dụng chấm dứt công việc. Bạn có thể viết đoạn mã đơn giản sau đây vào hàm thụ lý tình huống **FileExit\_Clicked**:

```
// Hàm thụ lý tình huống File | Exit
private void FileExit_Clicked(object sender, EventArgs e)
{
    // ghi nhận vào Application log...
    EventLog log = new EventLog();
    log.Log = "Application";
    log.Source = "PopUpMenu C#.NET Ch 16";
    log.WriteEntry("Bồ ơi, ứng dụng đóng lại rồi!");
    log.Close();
}
```

```
// bây giờ đóng lại ứng dụng
this.Close();
}
```

Nếu bây giờ bạn quan sát Application log, bạn sẽ thấy mục vào được hiển thị trên hình 2-34 là đã được chèn vào.

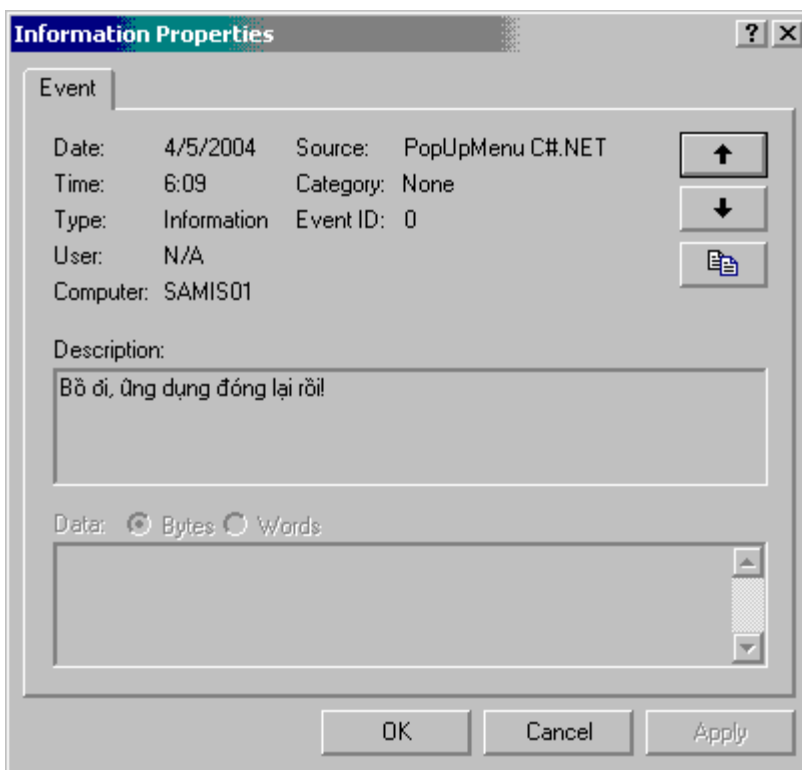


Hình 2-34: Application log custom của chúng ta

Muốn thấy thông điệp mục vào bạn chỉ cần double-click lên log entry thì thấy ngay, hình 2-35.

## 2.12.1 Đọc từ Event Log

Bây giờ bạn muốn đọc một vài thông tin từ một event log nào đó. Điều này không khó khăn gì lắm. Bạn nhớ lại, lớp **EventLog** có định nghĩa một thuộc tính mang tên **Entries**. Thuộc tính này trả về một thể hiện của lớp **EventLog.EventLogEntryCollection**. Collection này chứa một vài lớp indexable của **EventLogEntry**, mỗi lớp tượng trưng cho một mục vào trên một event log nào đó. Xem bảng 2-29.



Hình 2-35: Thông điệp của chúng ta

Bảng 2-29: Các thành viên của lớp *EventLogEntry*

Các thành viên	Ý nghĩa
<b>Category</b>	Đi lấy phần văn bản được gắn liền với CategoryNumber đối với mục vào này.
<b>CategoryNumber</b>	Đi lấy số category của ứng dụng cụ thể đối với mục vào này.
<b>Data</b>	Đi lấy dữ liệu nhị phân được gắn liền với mục vào này.
<b>EntryType</b>	Đi lấy kiểu dữ liệu của mục vào.
<b>EventID</b>	Đi lấy mã nhận diện event của mục vào này đối với một ứng dụng cụ thể.
<b>MachineName</b>	Đi lấy tên máy đã kết sinh ra mục vào này.

<b>Message</b>	Đi lấy thông điệp tương ứng với mục vào event này.
<b>Source</b>	Đi lấy tên ứng dụng đã kết sinh ra mục vào này.
<b>TimeGenerated</b>	Đi lấy thời gian kết sinh ra mục vào này.
<b>TimeWritten</b>	Đi lấy thời gian theo đây event được viết lên log, theo thời gian tại chỗ (local time).
<b>UserName</b>	Đi lấy tên người sử dụng chịu trách nhiệm đối với event này.

Nếu bạn cho nhậ tu hàm hành sự FileExit\_Clicked như sau:

```
private void FileExit_Clicked(object sender, EventArgs e)
{
    ...
    // cho hiển thị 5 entry trên Application log
    for(int i = 0; i < 5; i++)
    {
        try
        {
            MessageBox.Show("Message: " + log.Entries[i].Message + "\n" +
                            "Box name: " + log.Entries[i].MachineName +
                            "\n" + App: " + log.Entries[i].Source + "\n" +
                            "Time entered: " + log.Entries[i].TimeWritten,
                            "Application Log entry: ");
        }
        catch {}
    }
}
```



Bạn sẽ thấy “phụ” lên 5 thông điệp, tùy thuộc vào có gì thực thụ trên event log. Xem hình 2-36:

**Hình 2-36: Đọc Event Log**

## Chương 3

# Tìm hiểu về Assembly và cơ chế Version

Những ứng dụng mà bạn đã triển khai trong các chương đi trước của tập I bộ sách này, phần lớn đều là những ứng dụng cổ điển thuộc loại “đứng một mình” (stand alone), mang tính “tự cung tự cấp”, vì toàn bộ phần logic lập trình đều nằm gọn trong một tập tin .EXE duy nhất. Tập tin .EXE này bây giờ thường được gọi là “binary”, vì nó là một đối tượng ở dạng nhị phân.<sup>5</sup> Một khía cạnh trong lập trình trên sàn diễn .NET là .NET cung cấp cho bạn khả năng sử dụng lại (reuse) những binary này, khả năng truy cập những kiểu dữ liệu giữa các binary một cách hoàn toàn độc lập đối với ngôn ngữ lập trình. Nghĩa là trên .NET bạn có thể tạo một ứng dụng VB.NET trong ấy dẫn xuất (deriving) từ những lớp được viết theo C#, hoặc ngược lại. .NET gọi việc làm này là *cross-language inheritance* (kế thừa xuyên ngôn ngữ). Muốn hiểu điều này được thực hiện thế nào, đòi hỏi bạn tìm hiểu sâu về assembly.

Một khi bạn đã “thăm” về cách bố trí vật lý và logic của một assembly (và manifest liên đới) bạn phải biết thế nào là “private assembly” và “shared assembly”. Ngoài ra, bạn phải biết cách .NET Runtime giải quyết thế nào việc dò tìm nơi tá túc của một assembly và nhận biết ra vai trò của **Global Assembly Cache** (GAC). Gắn gũi với việc dò tìm nơi tá túc của assembly là khái niệm về tập tin cấu hình (\*.config) trong một ứng dụng. Như bạn có thể thấy, .NET Runtime có thể đọc dữ liệu của tập tin \*.config ở dạng XML để có thể gắn kết một phiên bản cụ thể nào đó đối với một shared assembly.

## 3.1 Tổng quan về .NET Assembly

Đơn vị cơ bản của lập trình .NET là *assembly*<sup>6</sup>. Một assembly là một binary mang một phiên bản, tự mô tả được (xuất hiện trước người sử dụng như là một .DLL hoặc một .EXE duy nhất) và chứa đựng một vài collection các kiểu dữ liệu (class, interface, structure, v.v..) cũng như một số nguồn lực tùy chọn (image, string table, v.v..). DLL (Dynamic Link Library) là những collection các lớp và hàm hành sự chỉ được kết nối với nhau trong chương trình đang chạy khi chúng được cần đến.

---

<sup>5</sup> Bạn chớ ngạc nhiên, bọn Mỹ khoái đặt từ mới cho những khái niệm cũ làm bạn lúng túng.

<sup>6</sup> Từ này nên hiểu như là “một dây chuyền lắp ráp” trong ngành may mặc hoặc da dầy, nên tạm thời chúng tôi không dịch, vì dịch ra là “lắp ráp” nghe nó vô duyên sao ấy.

Assembly được xem như là đơn vị .NET dùng lại được (“khả tái dụng”, reusable), mang một phiên bản nhất định, an toàn và có thể được triển khai dễ dàng. Chương này sẽ đi sâu vào assembly bao gồm kiến trúc và nội dung các assembly, assembly riêng tư (private assembly) và assembly được chia sẻ sử dụng (shared assembly).

Ngoài đoạn mã ứng dụng, assembly còn chứa những nguồn lực (resource<sup>7</sup>), chẳng hạn các tập tin gif, các định nghĩa kiểu dữ liệu đối với mỗi lớp bạn định nghĩa cũng như metadata liên quan đến đoạn mã và dữ liệu. Chương kế tiếp sẽ khảo sát metadata.

### 3.1.1 Các tập tin PE

Trên đĩa từ, các assembly thường là những tập tin PE (Portable Executable) và COFF (Common Object File Format). Các tập tin PE không phải mới mẻ gì. Dạng thức của một tập tin .NET PE giống đúc tập tin PE của Windows thông thường. Khác biệt rõ ràng là các tập tin PE/COFF cổ điển chứa những chỉ thị nhắm vào một sàn diễn đích và CPU cụ thể nào đó. Trong khi ấy, .NET binary thì chứa đoạn mã viết theo MSIL (Microsoft Intermediate Language, gọi tắt IL). Vào lúc chạy, đoạn mã IL sẽ được biên dịch bởi trình biên dịch JIT (just-in-time) dựa theo sàn diễn và CPU hiện hành.

Các tập tin PE được thi công như là những DLL hoặc EXE. Về mặt logic (tương phản với mặt vật lý), assembly thường bao gồm một hoặc nhiều *modules* (đơn nguyên). Tuy nhiên, bạn nên để ý là assembly chỉ có một điểm đột nhập (entry point) - **DLLMain**, **WinMain** hoặc **Main**. **DLLMain** là điểm đột nhập vào DLL, còn **WinMain** là điểm đột nhập vào ứng dụng Windows, và **Main** là điểm đột nhập vào các ứng dụng DOS hoặc Console.

Các modules được tạo ra như là những DLL và là thành phần kết cấu của một assembly. Nằm riêng một mình, module không thể thi hành được, nên phải được phối hợp thành assembly mới có thể hữu ích.

Bạn cho triển khai (deployment) và sử dụng lại toàn bộ nội dung của một assembly như là một đơn vị. Các assembly sẽ được nạp vào theo yêu cầu, và sẽ không được nạp vào nếu không thấy cần thiết.

### 3.1.2 Cấu trúc của một Assembly

#### *Assembly gồm một tập tin hoặc nhiều tập tin*

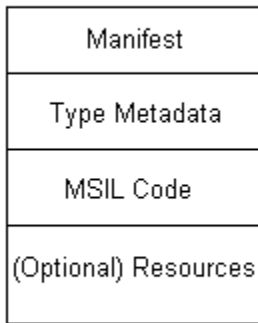
Một assembly thường bao gồm nhiều thành phần như theo hình 3-1:

---

<sup>7</sup> Chúng tôi không dịch là tài nguyên (giống như hàm mở, sông suối, v.v..).

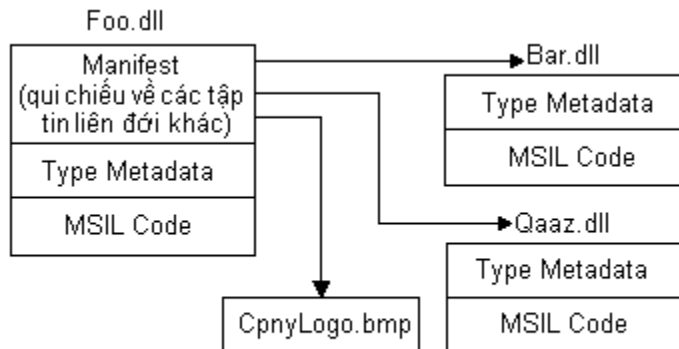
- Một **Manifest** (còn gọi là *assembly metadata*) mô tả trọn vẹn assembly.
- Một **Type Metadata** mô tả các kiểu dữ liệu và hàm hành sự được “xuất khẩu”.
- Một **MSIL Code**, là đoạn mã viết theo ngôn ngữ trung gian.
- Các **Resources** (tùy chọn) là những nguồn lực như hình ảnh, chuỗi, v.v..

#### Một File Assembly duy nhất - Foo.dll



**Hình 3-1: Một Single File Assembly**

Hình 3-01 cho thấy một assembly chỉ gồm một tập tin duy nhất (single file assembly). Theo hình này tất cả manifest, type metadata, MSIL code và resources đều nằm trọn trong một tập tin. Người ta còn gọi assembly này chỉ gồm có một đơn nguyên (module). Một module chẳng qua chỉ là một tên chung đối với một tập tin hợp lệ. Theo chiều hướng này, một assembly có thể được xem như là một đơn vị triển khai (thường được gọi là một “logical DLL”). Trong nhiều trường hợp, một assembly thật ra được hình thành bởi một module duy nhất. Trong trường hợp này, có một tương ứng một-đối-một giữa assembly (logic) và binary (vật lý) nằm đằng sau, như theo hình 3-1.



**Hình 3-2: Một multi-module assembly**

Tuy nhiên, một assembly cũng có thể trải dài trên nhiều tập tin, mà ta gọi là multifile assembly hoặc multi-module assembly, như theo hình 3-2. Theo hình 3-2 này, assembly được trải dài trên 4 tập tin: Foo.dll chứa đựng manifest (hoặc assembly metadata), Type metadata và MSIL Code nhưng lại không chứa Resources. Assembly sử dụng một tập tin hình ảnh CpnyLogo.bmp không nằm trong lòng Foo.dll, nhưng được qui chiếu trong lòng manifest. Assembly meta data còn qui chiếu hai module .DLL (Bar.dll và Qaaz.dll), mỗi module chỉ gồm Type Metadata và MSIL Code. Như thế một module sẽ không có assembly metadata, không có thông tin về phiên bản và không thể được cài đặt riêng rẽ.

Tất cả 4 tập tin vừa kể trên hình thành một assembly duy nhất.

### 3.1.3 Metadata là gì?

*Metadata*<sup>8</sup> là thông tin được trữ trên assembly mô tả những kiểu dữ liệu và những hàm hành sự thuộc assembly và cung cấp những thông tin hữu ích khác liên quan đến assembly. Thí dụ, nếu bạn tạo một lớp mang tên **JoyStick** sử dụng một ngôn ngữ “ăn ý” với .NET, thì trình biên dịch tương ứng sẽ tạo ra những metadata mô tả tất cả các vùng mục tin, hàm hành sự, thuộc tính và tính hướng định nghĩa bởi kiểu dữ liệu này. .NET Runtime sẽ dùng metadata này để giải quyết việc xác định nơi tá túc của các kiểu dữ liệu (với các thành viên của lớp) trong lòng binary, tạo những thể hiện cũng như giải quyết việc triệu gọi hàm từ xa.

Chính nhờ metadata, nên các assembly thường được gọi là *self-describing* (tự mình mô tả lấy), vì metadata mô tả một cách trọn vẹn mỗi module. Chương kế tiếp sẽ đi sâu vào metadata.

### 3.1.4 Assembly Manifest

Mỗi assembly đều có một *manifest*<sup>9</sup> được gắn liền (còn được gọi là “assembly metadata”) mô tả những gì được chứa đựng trong assembly, bao gồm:

- Tên nhận diện (**identity** name), phiên bản (**version**), nền văn hóa (**culture**), và mục khóa công cộng (**public key**).
- Một **danh sách các tập tin** thuộc assembly. Một assembly đơn lẻ phải có ít nhất một tập tin, nhưng có thể có một số tập tin nào đó.
- Một **danh sách các assembly được qui chiếu**. Tất cả các assembly khác được dùng bởi assembly này sẽ được sưu liệu trong manifest này, bao gồm thông tin liên quan đến số phiên bản, public key.
- Một bộ các **permission request** - những quyền hạn cần có để được phép chạy assembly này.
- Một **danh sách các kiểu dữ liệu và nguồn lực** trong assembly, một bản đồ (map) nối liền các kiểu dữ liệu public với các đoạn mã thi công.

Với cách này, một .NET assembly hoàn toàn tự mô tả lấy mình. Kể cả một chương trình đơn giản cũng có một manifest. Bạn có thể quan sát manifest bằng cách dùng trình

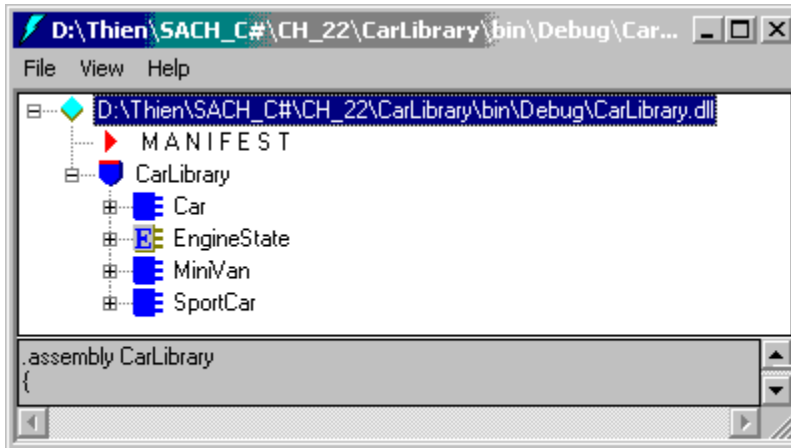
---

<sup>8</sup> Metadata là loại dữ liệu dùng mô tả những thuộc tính của các dữ liệu khác

<sup>9</sup> Từ này bắt chước ngành hàng không & hàng hải, theo đây manifest là không vận đơn hoặc hải vận đơn



tiện ích ILDasm, đi kèm theo IDE. Khi bạn cho mở một chương trình với trình tiện ích ILDasm, một chương trình .DLL sẽ giống như hình 3-3 và manifest giống như hình 3-4.



Hình 3-03: CarLibrary.dll được hiển thị bởi ILDasm.exe

assembly; đây là assembly được qui chiếu bởi tất cả các ứng dụng. Assembly **mscorlib** là thư viện cốt lõi đối với .NET và bao giờ cũng có sẵn trên mỗi sản phẩm .NET. Tiếp theo là qui chiếu về một assembly thứ hai về **System.Windows.Forms**.

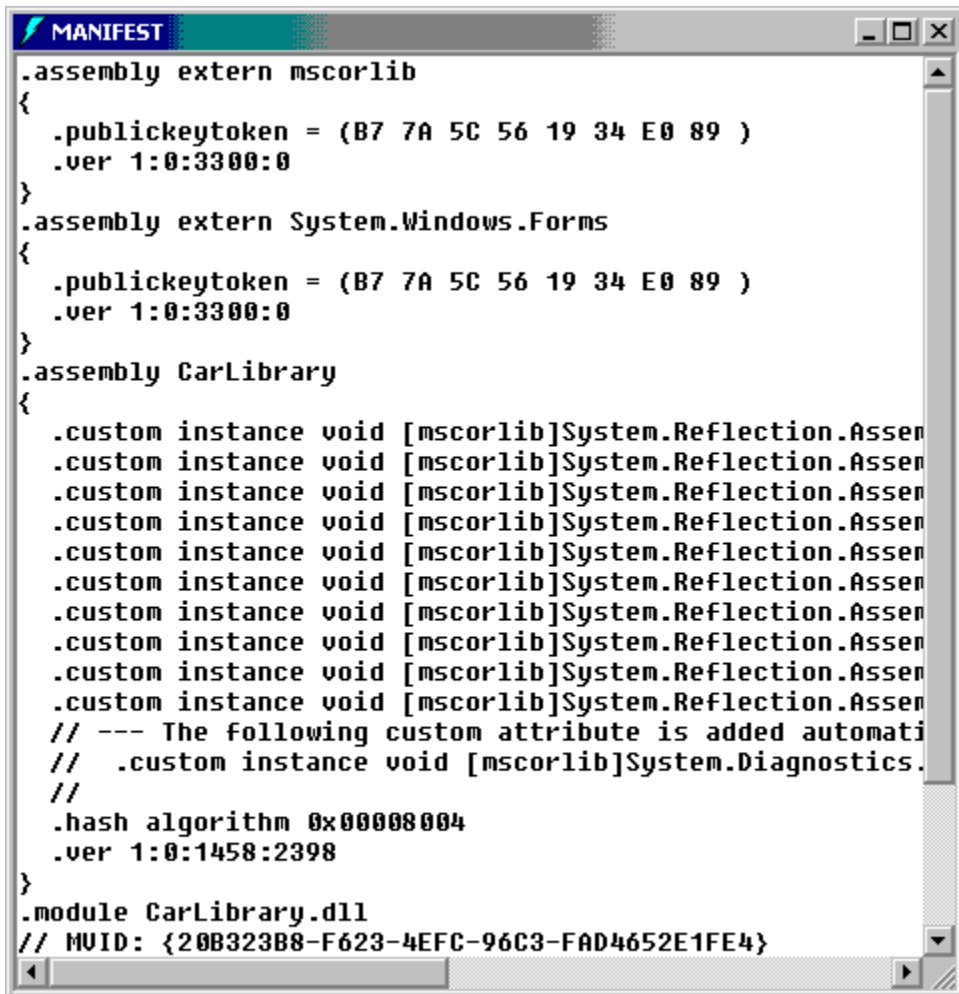
Dòng assembly kế tiếp là một qui chiếu về assembly **CarLibrary**. Bạn có thể thấy là assembly này chỉ có một module duy nhất. Tạm thời, bạn khỏi quan tâm đến phần còn lại của metadata.

### 3.1.4.1 Các module trên Manifest

Một assembly nào đó có thể có một hoặc nhiều module. Một module trong thực tế chỉ là một tên chung chung đối với một tập tin hợp lệ. Theo cách nhìn này, một assembly có thể được xem như là một đơn vị thi công (thường được gọi là “logical DLL”). Trong nhiều trường hợp, một assembly thật ra chỉ gồm một module duy nhất. Trong trường hợp này, có một tương ứng một-đối-một giữa assembly (lôgic) và binary (vật lý) nằm đằng sau, như theo hình 3-1. Và cũng trong trường hợp này, manifest sẽ bao gồm một hash code nhận diện mỗi module bảo đảm là mỗi khi chương trình thi hành thì chỉ phiên bản thích ứng của module sẽ được nạp vào ký ức. Nếu một module nào đó có nhiều phiên bản khác nhau, thì hash code bảo đảm là chương trình của bạn sẽ được nạp vào một cách thích ứng. Hash code là một biểu diễn số của đoạn mã của module, và nếu đoạn mã thay đổi, thì hash code sẽ không khớp. Trên hình 3-4, hàng hash algorithm cho biết hash code.

Trên hình phía trên có dòng MANIFEST. Nếu bạn double-click lên dòng này, thì cửa sổ Manifest hiện lên là hình 3-4.

Manifest được xem như là bản đồ cho thấy nội dung của assembly. Bạn có thể thấy dòng đầu tiên là một qui chiếu về **mscorlib**



Hình 3-04: Nội dung Manifest của CarLibrary.dll

### 3.1.4.2 Module Manifest

Như theo hình 3-1, mỗi module sẽ có riêng cho mình một manifest được tách rời khỏi assembly manifest. Module manifest sẽ cho liệt kê tất cả các assembly được qui chiếu bởi module đặc biệt này. Ngoài ra, nếu module có khai báo bất cứ kiểu dữ liệu nào, Type Metadata, thì các kiểu dữ liệu này sẽ được liệt kê trong module manifest, kèm theo đoạn mã thi công module, MSIL Code. Một module cũng có thể chứa các nguồn lực, Optional Resources, chẳng hạn những tập tin hình ảnh mà module này sử dụng đến.

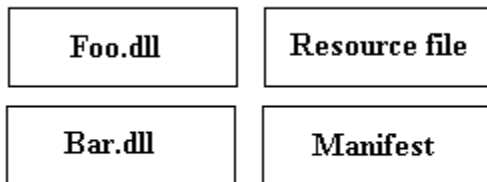
### 3.1.4.3 Các assembly khác được cần đến

Assembly manifest cũng chứa những qui chiếu về các assembly khác được yêu cầu. Mỗi qui chiếu như thế bao gồm tên assembly khác, số thứ tự phiên bản và culture cần thiết và originator của các assembly khác (tùy chọn). Originator là dấu ấn digital đối với nhà triển khai hoặc công ty cung cấp assembly khác,

**Bạn để ý:** Culture (văn hóa) là một đối tượng tượng trưng cho ngôn ngữ và những đặc tính hiển thị quốc gia đối với người sử dụng chương trình của bạn. Chính culture sẽ xác định chẳng hạn liệu xem dữ liệu ngày tháng năm được hiển thị theo dạng thức M/D/Y hoặc D/M/Y.

### 3.1.5 Hai cái nhìn của một assembly: vật lý và logic

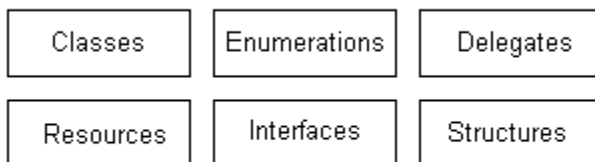
#### Cái nhìn vật lý của một Assembly



**Hình 3-05:** Về mặt vật lý, assembly là một collection các module.

Khi bạn bắt đầu làm việc với những .NET binary, bạn nên nhìn một assembly (single file hoặc multi file) theo hai cái nhìn ý niệm khác nhau. Khi bạn tạo dựng một assembly, bạn quan tâm đến cái nhìn vật lý (*physical view*). Xem hình 3-05. Trong trường hợp này, assembly có thể được thực hiện như là một số tập tin chứa các kiểu dữ liệu custom cũng như nguồn lực của bạn.

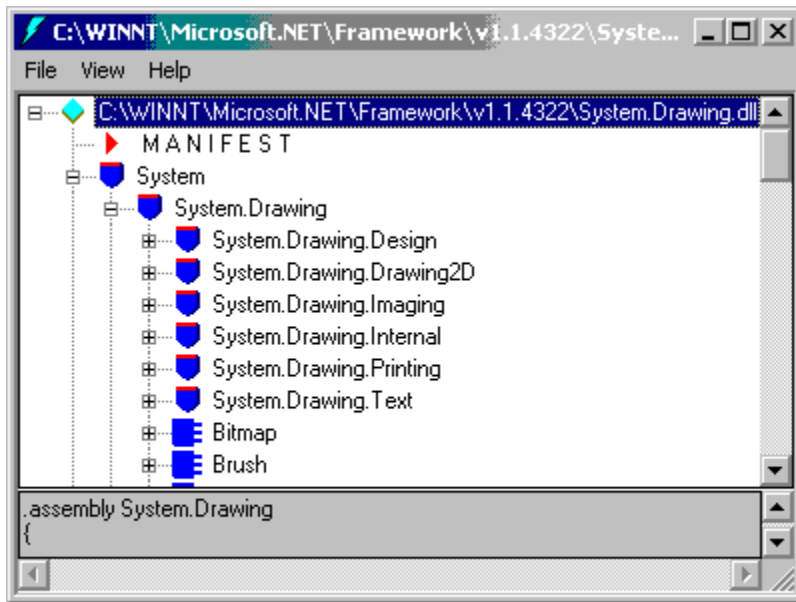
#### Cái nhìn lô gic của một assembly



**Hình 3-06:** Về mặt lô gic, assembly là một collection những kiểu dữ liệu.

Còn khi bạn là một người tiêu thụ assembly, thì bạn lại quan tâm đến cái nhìn lô gic (*logical view*). Xem hình 3-06. Trong trường hợp này, bạn có thể hiểu một assembly như là một collection các kiểu dữ liệu public, được đánh dấu phiên bản, mà bạn có thể sử dụng trong ứng dụng hiện hành của bạn. Thí dụ, assembly

**System.Drawing.dll** mà Microsoft đã tạo ra cho bạn về mặt vật lý được xem như là một binary DLL để bạn có thể tiêu thụ trong ứng dụng của bạn. Nhưng bạn có thể xem nó về mặt lô gic như là một collection những kiểu dữ liệu có liên hệ với nhau. Lẽ dĩ nhiên, ILDasm.exe là một công cụ rất hữu ích cho phép bạn khám phá cách bố trí lô gic của một assembly nào đó. Xem hình 3-07.



Hình 3-07: Logical View đối với System.Drawing.dll

### 3.1.6 Assembly khuyến khích việc tái sử dụng đoạn mã

Assembly chứa đoạn mã được thi hành bởi .NET Runtime. Như bạn có thể tưởng tượng, các kiểu dữ liệu cũng như các nguồn lực được chứa trong một assembly có thể được chia sẻ sử dụng và tái sử dụng bởi nhiều ứng dụng khác nhau. Ta cũng có thể cấu hình hóa các **private assembly** (trong thực tế đây là chọn lựa mặc nhiên). **Private assembly** dành cho ứng dụng đơn lẻ chạy trên một máy tính nào đó. Bạn sẽ thấy là private assembly sẽ đơn giản hóa việc triển khai cũng như việc phiên bản hóa ứng dụng của bạn. Việc có thể tái sử dụng những đoạn mã trên sàn diễn .NET tăng cường khái niệm độc lập về ngôn ngữ lập trình. Khi một ngôn ngữ lập trình nào “ăn ý” (aware) với .NET và tuân thủ các quy tắc do CLS (Common Language Specification) đề ra thì việc lựa chọn viết theo một ngôn ngữ lập trình lúc ấy là tùy sở thích của bạn.

Do đó, ta không những có khả năng sử dụng lại những kiểu dữ liệu giữa các ngôn ngữ lập trình mà còn nói rộng các kiểu dữ liệu xuyên qua các ngôn ngữ. Một ứng dụng viết theo C# có thể cho dẫn xuất từ một lớp được viết theo VB .NET chẳng hạn, hoặc ngược lại. Ta gọi là cross-language inheritance.

### 3.1.7 Assembly thiết lập Type Boundary và Security Boundary

Các assembly thiết lập **security boundary** (biên giới an toàn) cũng như **type boundary** (biên giới kiểu dữ liệu). Nghĩa là, một assembly được xem như là scope boundary (biên giới của một phạm trù) đối với những kiểu dữ liệu mà assembly chứa đựng. Lẽ dĩ nhiên là bạn có thể qui chiếu những kiểu dữ liệu xuyên biên giới assembly bằng cách thêm một reference về assembly cần thiết, hoặc trên IDE hoặc trên command line vào lúc biên dịch. Điều bạn không thể làm được là định nghĩa một kiểu dữ liệu nằm “chàng hảng” trên hai assembly khác nhau. Nói cách khác, assembly thường được dùng để định nghĩa một biên giới đối với những kiểu dữ liệu (và nguồn lực) mà assembly chứa đựng. Trên .NET, “lý lịch” của một kiểu dữ liệu nào đó sẽ được định nghĩa bởi assembly mà nó tá túc. Do đó, nếu hai assembly đều định nghĩa một kiểu dữ liệu (lớp, struct, hoặc gì gì đó) cùng mang tên giống nhau, đều được xem là không “dây mơ rễ má” gì cả, là những thực thể độc lập trong thế giới .NET.

Ngoài ra, một assembly cũng có thể thông tin an toàn (security information). Theo .NET Runtime, mức độ an toàn được đánh giá ở cấp assembly. Thí dụ, nếu Assembly A muốn sử dụng một lớp thuộc Assembly B, thì Assembly B sẽ là thực thể được chọn cung cấp truy cập (hoặc không). Các hạn buộc về an toàn (security constraint) được định nghĩa trong assembly sẽ được liệt kê một cách tường minh trong lòng manifest. Bạn nên nhớ là việc truy cập vào nội dung của một assembly sẽ được kiểm tra bằng cách sử dụng assembly metadata.

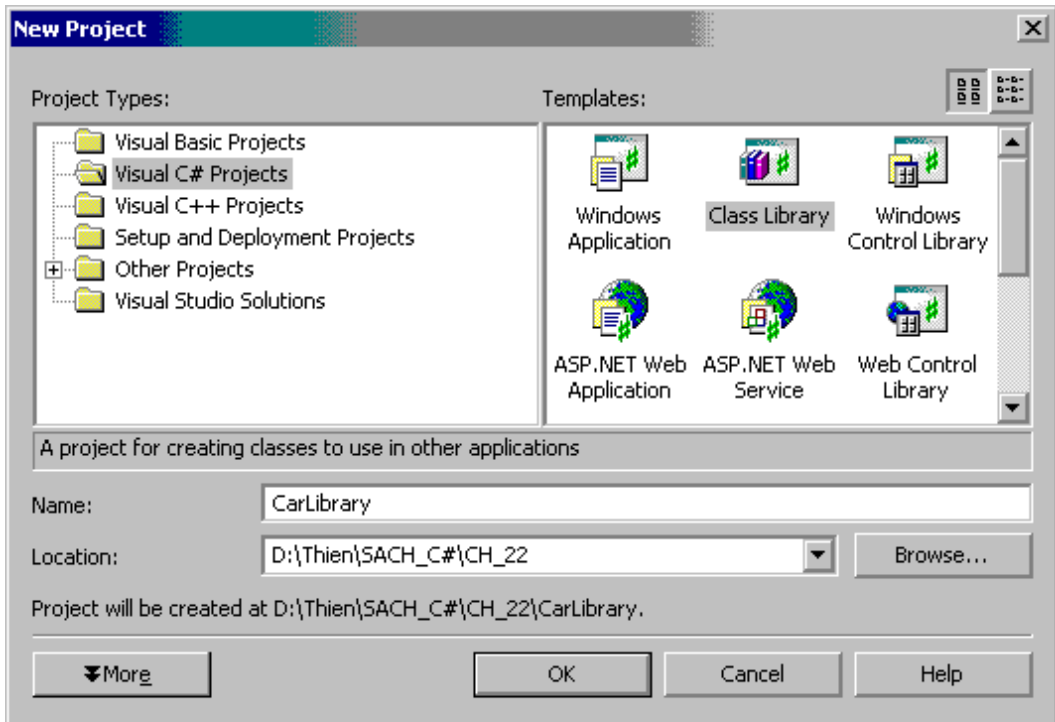
### 3.1.8 Phiên bản hóa một assembly

Mỗi assembly đều mang một mã nhận diện số phiên bản (gọi là version identifier) được áp dụng đối với tất cả các kiểu dữ liệu cũng như các nguồn lực được chứa đựng trong mỗi module của assembly. Mỗi phiên bản chỉ qui chiếu về nội dung của một assembly đơn lẻ. Tất cả các kiểu dữ liệu cũng như nguồn lực trong lòng assembly thay đổi cùng lúc với phiên bản. Sử dụng đến một version identifier, CLR có khả năng nạp đúng assembly mà phía ứng dụng client yêu cầu. Một version identifier thường gồm hai phần cơ bản: một chuỗi văn bản mang tính “thân thiện” (được gọi là *informational version*) và một mã số (mã nhận diện tương thích kiểu số, được gọi là *compatibility version*).

Thí dụ, giả sử bạn đã tạo một assembly mới với một chuỗi informational là “MyInterestingTypes”. Assembly này còn định nghĩa một dãy số compatibility chẳng hạn **1:0:70:3**. Con số compatibility version bao giờ cũng mang dạng thức tổng quát (gồm 4 số phân cách bởi dấu hai chấm). Hai số đầu nhận diện major version và minor version của assembly (trong trường hợp này là **1:0**). Còn số thứ ba (ở đây là **70**) cho biết build number, theo sau là phiên bản hiện hành (ở đây là 3).

## 3.2 Thử xây dựng một Single File Test Assembly

Bây giờ ta thử xây dựng một code library (đoạn mã thư viện) tối thiểu và trọn vẹn sử dụng C#. Về mặt vật lý đây là một assembly chỉ có một tập tin duy nhất, và ta cho mang tên **CarLibrary**.



**Hình 3-08: Chọn một workspace cho dự án Class Library**

Ta sử dụng IDE của Visual Studio .NET để tạo code library này. Bạn ra lệnh **File | New | Project** rồi chọn mục **Class Library** trên khung **Template**, và **Visual C#** trên khung **Project Types**. Trên ô **Name** bạn gõ vào **CarLibrary**, rồi gõ tiếp tên thư mục, cuối cùng ấn <OK>. Xem hình 3-8. Thí dụ 3-1 cho thấy bảng liệt in lớp CarLibrary.

Việc thiết kế thư viện xe hơi của chúng tôi bắt đầu với một lớp cơ bản trừu tượng mang tên **Car**; lớp này định nghĩa một số thành viên dữ liệu protected trưng ra những thuộc tính “cây nhà lá vườn” (custom properties). Lớp này chỉ có một hàm hành sự trừu tượng duy nhất mang tên **TurboBoost()**, và sử dụng một Enumeration duy nhất mang tên **EngineState**. Sau đây là định nghĩa ban đầu của namespace CarLibrary.

**Thí dụ 3-01: Bảng liệt in lớp CarLibrary**

```

*****
using System;
// Code library đầu tiên của ta (CarLibrary.dll)
namespace CarLibrary
{
    public enum EngineState //Cho biết tình trạng máy chạy hoặc chết
    {
        engineAlive,
        engineDead
    }

    public abstract class Car // Lớp cơ bản abstract
    {
        // Protected state data
        protected string petName; // tên cục cưng
        protected short currSpeed; // tốc độ hiện hành
        protected short maxSpeed; // tốc độ tối đa
        protected EngineState egnState; // tình trạng xe

        // hai hàm constructor
        public Car(){egnState = EngineState.engineAlive;}

        public Car(string name, short max, short curr)
        {
            egnState = EngineState.engineAlive;
            petName = name;
            maxSpeed = max;
            currSpeed = curr;
        }

        public string PetName
        {
            get { return petName; }
            set { petName = value; }
        }

        public short CurrSpeed
        {
            get { return currSpeed; }
            set { currSpeed = value; }
        }

        public short MaxSpeed
        {
            get { return maxSpeed; }
        }

        public EngineState EngineState
        {
            get { return egnState; }
        }

        public abstract void TurboBoost();
    }
}
*****

```

Bây giờ giả sử bạn có hai hậu duệ trực tiếp của kiểu dữ liệu **Car**, mang tên **MiniVan** và **SportCar**. Mỗi hậu duệ sẽ thi công hàm hành sự abstract **TurboBoost()** theo thể thức thích ứng. Thí dụ 3-02 là bảng liệt in hai lớp được dẫn xuất từ lớp **Car**.

**Thí dụ 3-02: Bảng liệt in hai lớp dẫn xuất từ lớp Car**

```

*****
using System;
using System.Windows.Forms;
// Code library đầu tiên của ta (CarLibrary.dll)
namespace CarLibrary
{
    . . .
    // Lớp SportCar được dẫn xuất từ lớp abstract Car
    public class SportCar: Car
    {
        // Hàm constructor
        public SportCar() {}
        public SportCar(string name, short max, short curr)
                                : base(name, max, curr) {}

        // Thi công hàm TurboBoost()
        public override void TurboBoost()
        {
            MessageBox.Show("Raming speed!", "Faster is better...");
        }
    }

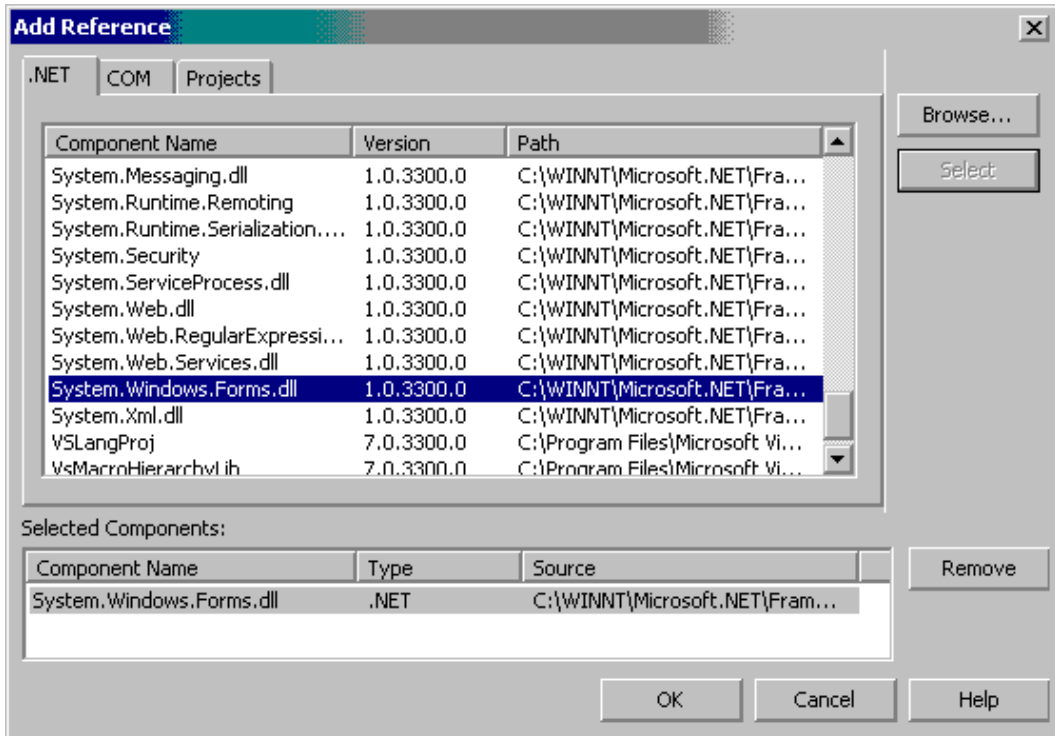
    // Lớp MiniVan được dẫn xuất từ lớp abstract Car
    public class MiniVan: Car
    {
        // Hàm constructor
        public MiniVan() {}
        public MiniVan(string name, short max, short curr)
                                : base(name, max, curr) {}

        // Thi công hàm TurboBoost()
        public override void TurboBoost()
        {
            // Khả năng turbo của minivan tối
            egnState = EngineState.engineDead;
            MessageBox.Show("Time to call AAA", "Your car is dead");
        }
    }
}
*****

```

Bây giờ bạn để ý cách các lớp dẫn xuất thi công hàm hành sự **TurboBoost()** sử dụng đến lớp **MessageBox**, được định nghĩa bởi assembly **System.Windows.Forms.dll**. Để assembly của bạn có thể sử dụng các kiểu dữ liệu được định nghĩa trong assembly này, dự án **CarLibrary** phải cho qui chiếu về **System.Windows.Forms.dll**. Muốn thế, trên dự án **CarLibrary**, bạn ra lệnh **Project | Add Reference...** để cho hiện lên khung đối thoại **Add Reference**. Bạn chọn Tab **.NET** trên khung đối thoại này rồi rảo tìm chọn ra **System.Windows.Form.dll**, ấn nút <Select>. Lúc này **System.Windows.Form.dll** hiện lên ở dưới tại khung **Selected Components**. Cuối cùng bạn ấn nút <OK>. Xem hình 3-9.





Hình 3-09: Qui chiếu về một assembly ngoài lại.

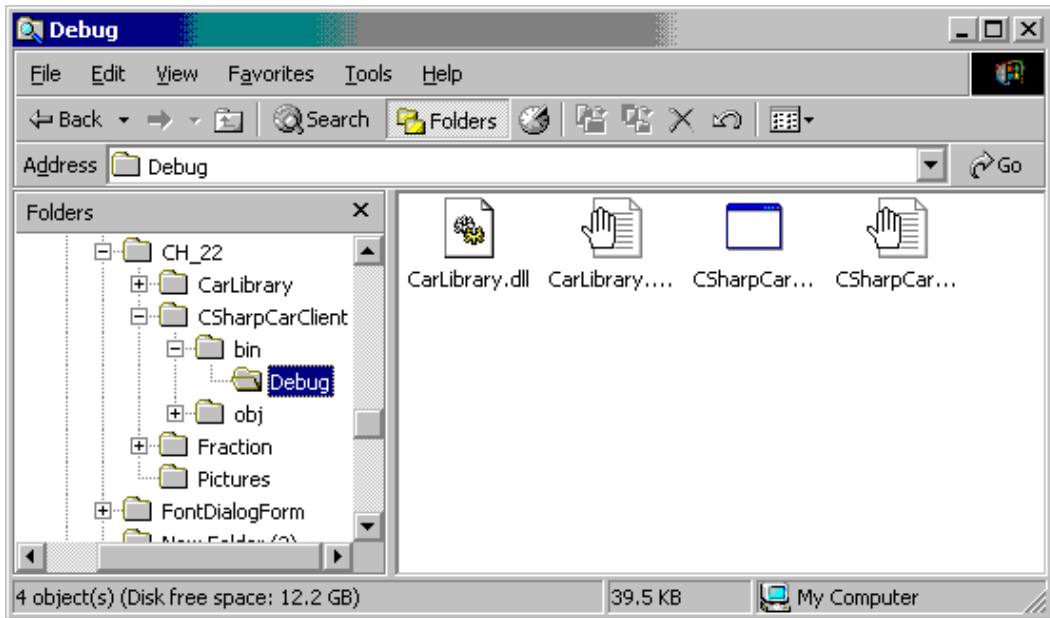
Như bạn có thể thấy về sau, namespace **System.Windows.Forms** chứa vô số kiểu dữ liệu cho phép bạn tạo dựng những ứng dụng GUI. Hiện thời thì chỉ lớp **MessageBox** là bạn quan tâm đến. Bây giờ bạn có thể **Build** dự án **CarLibrary** và bạn có một code library mới.

### 3.2.1 Một ứng dụng client C#: CSharpCarClient

Vì các lớp xe hơi mà chúng ta vừa tạo dựng ra được khai báo là “public” (sở hữu công cộng) nên các binary khác có khả năng sử dụng các lớp custom của ta. Bây giờ ta thử tạo một ứng dụng client sử dụng C#. Ta tạo một dự án C# Console Application. Bạn ra lệnh **File | New | Project**, rồi chọn **Visual C#** và **Console Application**, đặt tên cho dự án là **CSharpCarClient**, rồi ấn OK. Lúc này khung sườn dự án C# Console Application được hình thành.

Bước kế tiếp là cho đặt để một qui chiếu về CarLibrary.dll. Trên dự án **CSharpCarClient**, bạn ra lệnh **Project | Add Reference**, để cho hiện lên khung đối thoại **Add Reference**, bạn chọn **Tab Project**, rồi dùng nút **Browse** để tìm về thư mục của **CarLibrary.dll**. Khi tìm ra, bạn ấn nút **<Select>**, CarLibrary.dll xuất hiện trong khung **Selected Components**. Bạn ấn OK. Coi như qui chiếu về **CarLibrary.dll** được thiết lập.

Một khi bạn thêm một qui chiếu về assembly CarLibrary, Visual Studio .NET IDE sẽ cho chép assembly được qui chiếu và đưa nó vào thư mục Debug. Xem hình 3-10:



**Hình 3-10: Các bản sao cục bộ của các assembly được qui chiếu sẽ được ghi trên thư mục Debug**

Bây giờ xem như ứng dụng client được cấu hình sử dụng assembly CarLibrary, bạn tự do tạo một lớp khác sử dụng đến các kiểu dữ liệu của assembly CarLibrary.dll. Thí dụ 3-03 sau đây là một test driver.

**Thí dụ 3-03: Bảng liệt in ứng dụng CSharpCarClient sử dụng assembly CarLibrary**

```
*****
// Sử dụng đầu tiên một binary
using System;

namespace CSharpCarClient
{
    using CarLibrary;

    public class CarClient
    {
        [STAThread]
        public static int Main(string[] args)
        {
            // Tạo một xe sport
            SportCar viper = new SportCar("Viper", 240, 40);
            viper.TurboBoost();
        }
    }
}
```

```

        // Tạo một minivan
        MiniVan mv = new MiniVan();
        mv.TurboBoost();
        return 0;
    }
}
}
}
*****

```

Bạn thấy là ứng dụng client CSharpCarClient này sử dụng các kiểu dữ liệu được định nghĩa trong lòng một assembly duy nhất. Bạn cho chạy thử chương trình bạn thấy là có hai message box.

### 3.2.2 Một ứng dụng client VB.NET: VBCarClient

Khi bạn cài đặt Visual Studio .NET, bạn nhận được 4 ngôn ngữ lập trình có khả năng xây dựng những managed code: JScript.NET, C++ với managed extension (MC++), C#, và VB.NET. Tất cả các ngôn ngữ này đều dùng chung một IDE. Do đó, các lập trình viên VB.NET, ATL, C# và MFC đều dùng chung một môi trường triển khai phần mềm. Như thế, tiến trình xây dựng một ứng dụng VB.NET sử dụng CarLibrary rất đơn giản. Ta thử tạo một dự án ứng dụng VB.NET mới cho mang tên **VBCarClient**, bằng cách chọn **Visual Basic Projects** trên khung Project Types và **Windows Application** trên khung Template của khung đối thoại **New Project**.

Sau khi IDE kết sinh đoạn mã VB.NET, bạn cho đặt để một qui chiếu về **C# CarLibrary** bằng cách dùng khung đối thoại **Add Reference**. Giống như với C#, VB.NET đòi hỏi bạn liệt kê mỗi namespace được dùng trong dự án v.v.. Tuy nhiên, VB.NET sẽ sử dụng từ chốt “imports” thay vì chỉ thị “using” của C#. Do đó, bạn cho mở code window đối với Form và thêm các lệnh sau đây:

```

` Giống như C#, VB.NET cần xem các namespace mà một lớp sử dụng
Imports System
Imports System.Collections
. . .
Imports CarLibrary

```

Bạn sử dụng Designer để tạo một giao diện UI rất đơn giản làm việc với kiểu dữ liệu xe hơi. Xem hình 3-11. Hai nút là đủ.

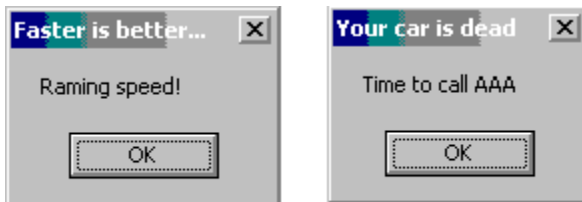


Hình 3-11: Một UI rất đơn giản

Bước kế tiếp là thêm các hàm thụ lý tình huống Click đối với mỗi nút. Muốn thế, khi đang ở chế độ design, bạn double click lên mỗi nút thì IDE sẽ kết sinh một đoạn mã khung sườn hàm thụ lý tình huống Click. Sau đây là đoạn mã;

```
Private Sub btnCar_Click(ByVal sender As System.Object,
    ByVal e As System.EventArgs) Handles btnCar.Click
    Dim sc As New SportCar()
    sc.TurboBoost()
End Sub

Private Sub btnMiniVan_Click(ByVal sender As System.Object,
    ByVal e As System.EventArgs) Handles btnMiniVan.Click
    Dim sc As New MiniVan()
    sc.TurboBoost()
End Sub
```



Hình 3-12: Kết quả chạy VBCarClient

Bạn để ý là mỗi subclass của Car sẽ được tạo sử dụng từ chốt New. Tuy nhiên, khác với VB 6.0, các lớp ở đây có những hàm constructor thật sự. Do đó, các dấu ngoặc rỗng trên tên lớp sẽ triệu gọi hàm constructor đối với lớp. Như bạn có thể chờ đợi, khi bạn cho chạy chương trình, mỗi

loại xe sẽ đáp ứng một cách thích ứng (hình 3-12).

### 3.2.3 Kế thừa xuyên ngôn ngữ lập trình

Một khái niệm khá lý thú trong triển khai .NET là tính kế thừa xuyên ngôn ngữ lập trình (cross-language inheritance). Để minh họa khái niệm này, ta thử tạo một lớp VB.NET mới được dẫn xuất từ CarLibrary.SportCar được viết theo C#. Có thể bạn cho là khó lòng thực hiện được. Thật thế, nếu bạn sử dụng Visual Basic 6.0. Tuy nhiên, với VB.NET, lập trình viên có khả năng sử dụng những tính năng thiên đối tượng có thể tìm thấy trong C#, Java và C++, kể cả kế thừa cô điển (nghĩa là mối liên hệ “is-a”).

Để minh họa, bạn thêm một lớp mới mang tên **PerformanceCar** vào ứng dụng client **VBCarClient**, bằng cách ra lệnh **Project | Add Class**. Trong đoạn mã theo sau, bạn để ý bạn đang dẫn xuất từ lớp C# Car sử dụng từ chốt VB.NET “Inherits”. Như bạn còn nhớ,

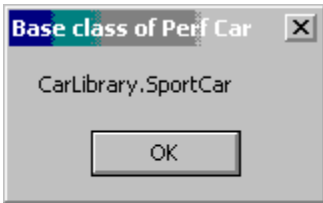
lớp Car định nghĩa một hàm hành sự abstract TurboBoost(), mà chúng tôi thi công bằng cách sử dụng từ chốt VB.NET “Overrides”:

```
'Vâng, VB.NET hỗ trợ lập trình thiên đối tượng
Imports CarLibrary
Imports System.Windows.Forms

'Lớp VB này được dẫn xuất từ C# SportCar
Public Class PerformanceCar
    Inherits CarLibrary.SportCar
    // Thi công hàm hành sự abstract TurboBoost
    Overrides Sub TurboBoost()
        MessageBox.Show("Blistering speed", "VB PerformanceCar says")
    End Sub
End Class
```

Bây giờ bạn thêm một button mang tên btnPreCar, rồi thêm đoạn mã sau đây:

```
Private Sub btnPerCar_Click(ByVal sender As System.Object,
    ByVal e As System.EventArgs) Handles btnPerCar.Click
    Dim pc As New PerformanceCar()
    pc.PetName = "Hank"
    MessageBox.Show(pc.GetType().BaseType.
        ToString(), "Base class of Perf Car")
    pc.TurboBoost()
End Sub
```



Hình 3-13: Kế thừa xuyên ngôn ngữ

Bạn để ý là ta có khả năng nhận diện lớp cơ sở về mặt lập trình (hình 3-13).

Tới đây, bạn để bắt đầu tiến trình chặt ứng dụng thành những khối xây dựng binary riêng rẽ. Dựa theo bản chất độc lập với ngôn ngữ của .NET, bất cứ ngôn ngữ nào nhắm tới vào vào lúc chạy sẽ có khả năng (và nói rộng) những kiểu dữ liệu được mô tả trong lòng một assembly nào đó.

### 3.2.4 Thử khảo sát manifest của CarLibrary.dll

Tới đây bạn đã thành công trong việc tạo một assembly chỉ gồm duy nhất một tập tin, **CarLibrary.dll** và hai ứng dụng client C#, **CSharpCarClient.exe**, và **VBCarClient.exe**. Bước kế tiếp là tìm hiểu sâu assembly .NET được xây dựng thế nào sau hậu trường. Bạn nhớ lại là mỗi assembly bao giờ cũng chứa một manifest liên đới, gọi là assembly manifest. Manifest thường chứa metadata khai báo cho biết tên và phiên bản của assembly, cũng như danh sách các module nội tại và ngoại lai cấu thành assembly. Ngoài ra, manifest còn thông tin về culture (trong việc phổ biến phần mềm cho cả thế giới), một “strong name” tương ứng (do “shared assembly” đòi hỏi, mà ta sẽ xem sau) và tùy chọn những thông tin về vấn đề an ninh và nguồn lực.

Các trình biên dịch “ăn ý” (aware) với .NET (chẳng hạn csc.exe) tự động tạo ra một manifest vào lúc biên dịch. Bây giờ bạn thử dùng trình tiện ích **ILDasm.exe**, xem **CarLibrary.dll** ra thế nào. Trình tiện ích này đọc metadata để cho hiển thị thông tin cần thiết đối với mỗi kiểu dữ liệu. Bạn xem lại hình 3-03 (trang 5).

Bây giờ, bạn cho mở manifest bằng cách double-click lên icon MANIFEST trên hình 3-03. Cửa sổ MANIFEST hiện lên như theo hình 3-04 (trang 6).

Khởi mã đầu tiên trên manifest khai báo cho biết tất cả những assembly nằm ngoài, mà assembly hiện hành cần đến để có thể hoạt động được. Như bạn có thể thấy, **CarLibrary.dll** cần đến các assembly **mscorlib.dll** và **System.Windows.Forms.dll**, nằm ở ngoài, mỗi assembly này được đánh dấu trên manifest bởi tag **[.assembly extern]**.

```
.assembly extern mscorlib
{
    .publickeytoken = (B7 7A 5C 56 19 34 E0 89 )    // .z\V.4..
    .ver 1:0:3300:0
}
.assembly extern System.Windows.Forms
{
    .publickeytoken = (B7 7A 5C 56 19 34 E0 89 )    // .z\V.4..
    .ver 1:0:3300:0
}
```

Ở đây, mỗi khối **[.assembly extern]** được khai báo bởi các chỉ thị **[.publickeytoken]** và **[.ver]**. Chỉ thị **[.publickeytoken]** chỉ hiện diện nếu assembly được cấu hình như là một shared assembly và được dùng qui chiếu “strong name” của shared assembly (sẽ được đề cập chi tiết về sau). Còn chỉ thị **[.ver]** cho biết mã nhận diện phiên bản kiểu số.

Sau phần khai báo các assembly nằm ngoài, sau đó manifest sẽ liệt kê mỗi module được chứa đựng trong assembly. Vì **CarLibrary.dll** là một single file assembly nên bạn chỉ tìm thấy một **[.module]** tag. Manifest này còn liệt kê một số attributes (được đánh dấu bởi tag **[.custom]**) chẳng hạn tên công ty, tên thương hiệu, v.v.. tất cả hiện trống rỗng.

```
.assembly CarLibrary
{
    .custom instance void [mscorlib]
System.Reflection.AssemblyKeyNameAttribute::.ctor(string)=(01 00 00 00
00)
    .custom instance void [mscorlib]
System.Reflection.AssemblyKeyFileAttribute::.ctor(string)=(0100 00 00 00)
    .custom instance void [mscorlib]
System.Reflection.AssemblyDelaySignAttribute::.ctor(bool)=(01 00 00 00
00)
    .custom instance void [mscorlib]
System.Reflection.AssemblyTrademarkAttribute::.ctor(string) = ( 01 00 00
00 00 )
    .custom instance void [mscorlib]
```

```

System.Reflection.AssemblyCopyrightAttribute::.ctor(string) = ( 01 00 00
00 00 )
.custom instance void [mscorlib]
System.Reflection.AssemblyProductAttribute::.ctor(string)=(0100 00 00 00)
.custom instance void [mscorlib]
System.Reflection.AssemblyCompanyAttribute::.ctor(string)=(01 00 00 00
00)
.custom instance void [mscorlib]
System.Reflection.AssemblyConfigurationAttribute::.ctor(string) = ( 01 00
00 00 00 )
.custom instance void [mscorlib]
System.Reflection.AssemblyDescriptionAttribute::.ctor(string)=(0100000000
)
.custom instance void [mscorlib]
System.Reflection.AssemblyTitleAttribute::.ctor(string)=(01 00 00 00 00)
// --- The following custom attribute is added automatically, do not
// uncomment -----
// .custom instance void [mscorlib]
// System.Diagnostics.DebuggableAttribute::.ctor(bool,
// bool) = ( 01 00 01 01 00 00 )
.hash algorithm 0x00008004
.ver 1:0:1458:2398
}
.module CarLibrary.dll
// MVID: {20B323B8-F623-4EFC-96C3-FAD4652E1FE4}
.imagebase 0x11000000
.subsystem 0x00000003
.file alignment 512
.corflags 0x00000001
// Image base: 0x03260000

```

Ở đây, bạn có thể thấy tag [.assembly] dùng đánh dấu tên thân thiện của custom assembly (CarLibrary). Giống như với những khai báo đối với các assembly ngoại lai, tag [.ver] định nghĩa số phiên bản compatibility đối với assembly này, theo đây [.hash] đánh dấu đoạn mã băm (hash code) được kết sinh của tập tin. bạn để ý là assembly CarLibrary không định nghĩa một [.publickeytoken] tag, vì CarLibrary không được cấu hình là một shared assembly.

Bảng 3-01 sau đây tóm lược các tag được sử dụng trong manifest.

**Bảng 3-01: Các tag của Manifest**

Các tag	Mô tả
<b>.assembly</b>	Đánh dấu khai báo một assembly cho biết tập tin là một assembly.
<b>.file</b>	Đánh dấu các tập tin phụ thêm (extra) trong cùng assembly.
<b>.class extern</b>	Các lớp được xuất khẩu bởi assembly nhưng được khai báo trong một module khác.
<b>.exeloc</b>	Cho biết nơi tá túc của chương trình khả thi đối với assembly.
<b>.manifestres</b>	Cho biết các nguồn lực (nếu có) của manifest. Bạn sẽ thấy tag này hiện diện khi làm việc với đồ họa GDI+.

<b>.module</b>	Khai báo module, cho biết tập tin là một module (nghĩa là một .NET binary không mang manifest) và không phải là một assembly.
<b>.module extern</b>	Những module của assembly này chứa những item được qui chiếu trong module này.
<b>.assembly extern</b>	Qui chiếu assembly cho biết một assembly khác chứa những item được qui chiếu bởi module này.
<b>.publickey</b>	Chứa các bytes hiện thời của mục khóa public.
<b>.publickeytoken</b>	Chứa một token của mục khóa public hiện thời.

### 3.2.5 Khảo sát các kiểu dữ liệu của CarLibrary.dll

Bạn nhớ cho là assembly không chứa chỉ thị đặc trưng của sàn diễn mà chỉ là một ngôn ngữ trung gian (Intermediate Language, IL). Khi .NET Runtime nạp assembly vào ký ức, phần IL nằm đằng sau sẽ được biên dịch (thông qua trình biên dịch JIT) thành chỉ thị mà máy bạn đang dùng có thể hiểu được. Ngoài ra, ngoài phân đoạn mã IL thô và assembly manifest, assembly còn chứa metadata lo mô tả mỗi kiểu dữ liệu được chứa đựng trong một module nào đó.

Thí dụ, trên hình 3-03 (trang 5), nếu bạn bung nhánh SportCar rồi double-click lên hàm hành sự **TurboBoost()**, thì ILDasm.exe sẽ cho mở một cửa sổ cho thấy những chỉ thị thô IL, như theo hình 3-14 sau đây. Bạn để ý là trên cửa sổ này, tag `[.method]` cho phép nhận diện một hàm hành sự được định nghĩa bởi kiểu dữ liệu SportCar.

```

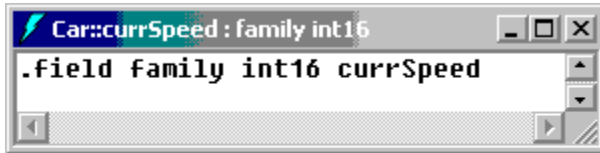
SportCar::TurboBoost : void()
.method public hidebysig virtual instance void
    TurboBoost() cil managed
{
    // Code size      17 (0x11)
    .maxstack 2
    IL_0000: ldstr      "Raming speed!"
    IL_0005: ldstr      "Faster is better..."
    IL_000a: call       valuetype [System.Windows.F
    IL_000f: pop
    IL_0010: ret
} // end of method SportCar::TurboBoost

```

Hình 3-14: IL đối với hàm SportCar.TurboBoost()



Như bạn có thể thấy, dữ liệu public được định nghĩa bởi một kiểu dữ liệu sẽ được đánh dấu bởi một tag `[.field]`, hình 3-15. Thí dụ, lớp **Car** đã định nghĩa một số vùng mục tin protected, chẳng hạn **currSpeed**. Nếu ta double-click vùng mục tin thì ILDasm.exe cho hiển thị một cửa sổ khác (hình 3-15). Bạn để ý, tag “family” cho biết là protected data.

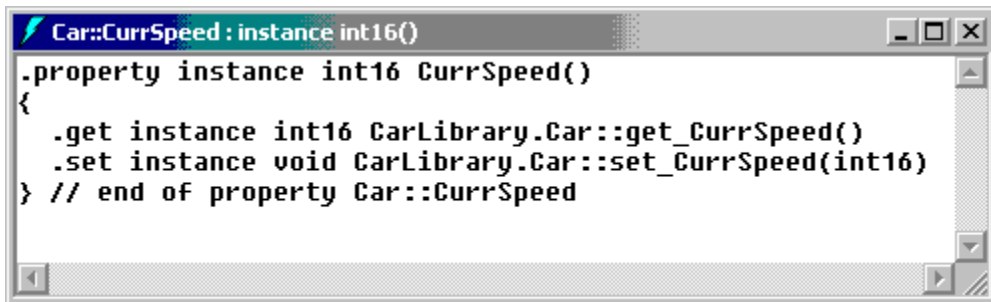


Hình 3-15: IL đối với vùng **currSpeed**

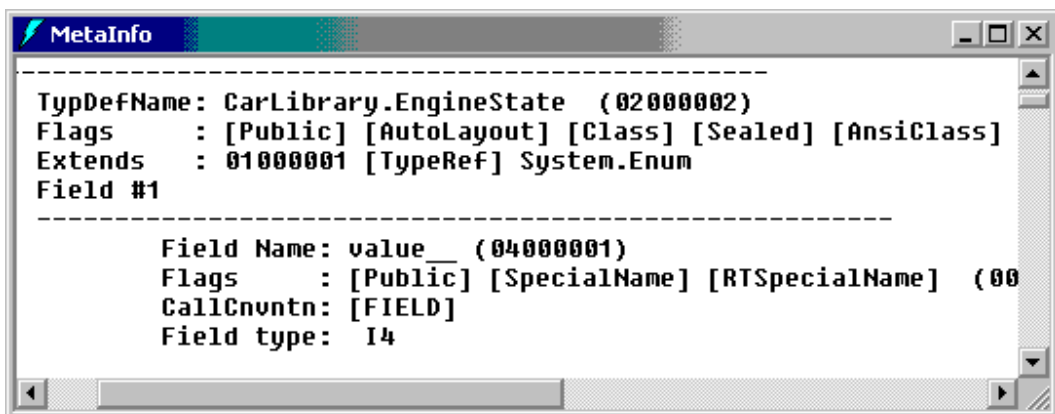
Ngoài ra, thuộc tính cũng được đánh dấu bởi tag `[.property]` như theo hình 3-16. Hình này cho thấy IL mô tả public property cho phép truy cập vùng mục tin **currSpeed** nằm đằng sau. Bạn để ý bản chất read/write của thuộc tính

**CurrSpeed** được đánh dấu bởi các tag `[.get]` và `[.set]`.

Bây giờ, nếu bạn khổ tổ hợp phím `<Ctrl + M>`, thì ILDasm.exe sẽ cho hiển thị cửa sổ MetaInfo cho biết metadata của mỗi kiểu dữ liệu (xem hình 3-17).



Hình 3-16: IL đối với thuộc tính **CurrSpeed**



Hình 3-17: Type metadata

Dựa trên metadata này, .NET Runtime có khả năng định vị và tạo dựng những thể hiện đối tượng và triệu gọi hàm hành sự. Nhiều công cụ khác nhau (chẳng hạn Visual Studio .NET) sử dụng metadata này vào lúc thiết kế để kiểm tra hợp lệ số thông số (cũng như kiểu dữ liệu thông số) khi đang biên dịch. Để tóm lược “câu chuyện”, bạn nên nắm vững các điểm sau đây:

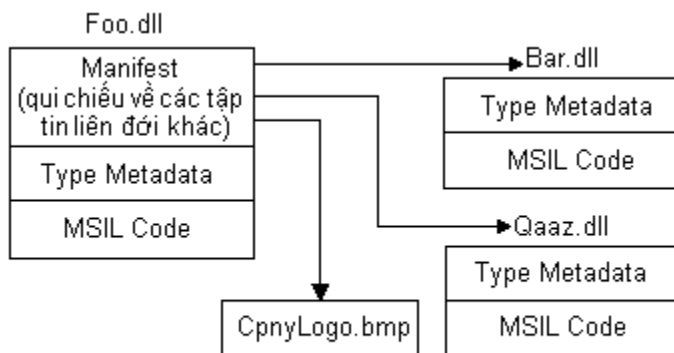
- Một assembly bao gồm một lô các module được đánh số phiên bản (versioned) và tự mô tả được (self-describing). Mỗi module sẽ chứa đựng một số kiểu dữ liệu và nguồn lực (tùy chọn).
- Mỗi assembly đều chứa đựng metadata lo mô tả tất cả các kiểu dữ liệu trong lòng một module nào đó. .NET Runtime (cũng như vô số công cụ thiết kế khác) sẽ dựa vào metadata này để tìm ra nơi tá túc của các đối tượng để có thể tạo ra đối tượng, kiểm tra hợp lệ các triệu gọi hàm hành sự, khởi động IntelliSense, v.v..
- Mỗi assembly đều chứa đựng một manifest liệt kê tất cả các tập tin nằm ngoài cũng như nằm trong mà assembly binary cần đến, các thông tin phiên bản cũng như các chi tiết khác liên quan đến assembly.

Tiếp theo, bạn phải phân biệt giữa private assembly và shared assembly. Chúng ta sẽ xem sau trong chốc lát.

### 3.3 Thử khảo sát Multi-Module Assembly

Một single-module assembly như trong thí dụ CarLibrary chỉ có một tập tin là một tập tin EXE hoặc DLL. Single module này chứa tất cả các kiểu dữ liệu và thi công đối với ứng dụng. Assembly manifest được đặt nằm lọt thỏm trong lòng module.

Còn multi-module assembly (còn gọi là Multifile assembly) thì lại bao gồm nhiều tập tin



**Hình 3-18: Một multi-module assembly**

(zero hoặc một EXE và zero hoặc nhiều tập tin DLL, mặc dù bạn phải có ít nhất một EXE hoặc DLL). Trong trường hợp này, assembly manifest có thể được nằm trong một tập tin đứng riêng, hoặc có thể được đặt nằm lọt thỏm trong lòng một trong những module. Khi assembly được qui chiếu, CLR sẽ nạp tập tin chứa manifest rồi sau đó mới nạp những

module cần thiết khi cần đến. Nói cách khác, multi-module assembly không nhất thiết được kết nối thành một tập tin đồ sộ. Thay vào đó, nó sẽ được kết nối thông qua thông tin được chứa trong assembly manifest duy nhất. Hình 3-18 (hình này giống đúc hình 3-3) cho thấy hình ảnh của một multi-module assembly.

### 3.3.1 Những lợi điểm khi dùng multi-module assembly

Multi-module assembly mang lại những lợi điểm đối với những chương trình thực tế, đặc biệt khi chương trình này được viết cùng lúc bởi nhiều lập trình viên, và chương trình rất đồ sộ.

Bạn thử tưởng tượng một dự án duy nhất sẽ được triển khai bởi 25 lập trình viên chẳng hạn. Nếu dự án này được tạo như là một single-module assembly và được trải nghiệm, thì cùng lúc 25 lập trình viên phải kiểm tra đoạn mã chót nhất của họ, và ứng dụng khổng lồ phải được xây dựng. Điều này sẽ gây ra một cơn ác mộng logic.

Tuy nhiên, nếu mỗi lập trình viên xây dựng riêng cho mình những module, thì chương trình có thể được xây dựng dựa trên module có sẵn chót nhất từ mỗi lập trình viên. Điều này sẽ gỡ bỏ vấn nạn “hậu cần” (logistic). Mỗi module có thể được kiểm tra khi đã sẵn sàng.

Có thể điểm quan trọng nhất là multi-module assembly cho phép triển khai dễ dàng hơn và có thể bảo trì một chương trình rộng lớn. Bạn thử tưởng tượng mỗi lập trình viên sẽ xây dựng riêng một module riêng rẽ cho mình, mỗi lập trình viên có riêng một DLL. Còn người chịu trách nhiệm xây dựng toàn bộ ứng dụng sẽ tạo một module thứ 26 với manifest đối với toàn bộ các assembly. Tất cả các 26 tập tin này sẽ được triển khai đối với người sử dụng. Người sử dụng chỉ cần nạp module có mang manifest, và trong lúc này không quan tâm đến 25 module kia. Manifest sẽ nhận diện một trong 25 module nào có chứa hàm hành sự nào mà người sử dụng cần đến và module thích ứng sẽ được nạp vào. Như vậy mọi việc sẽ trong suốt (transparent) đối với người sử dụng.

Khi các module được nhật tu, lập trình viên chỉ cần gởi đi các module được nhật tu (và một module với một manifest được nhật tu). Ta có thể bổ sung những module mới cũng như những module hiện hữu có thể bị gỡ bỏ; và người sử dụng tiếp tục chỉ nạp module với manifest.

Ngoài ra, không phải toàn bộ 25 module sẽ cần phải nạp vào chương trình. Bằng cách chẻ chương trình thành 25 module, bộ loader chỉ cần nạp những phần nào trong chương trình mà ta cần đến. Như vậy những module nào thì thoáng mới dùng đến thì chỉ được nạp vào khi cần đến, còn những module nào thường xuyên được dùng đến nhiều thì sẽ được nạp vào trong tình huống bình thường.

### 3.3.2 Xây dựng một multi-module assembly

Để minh họa việc sử dụng những multi-module assembly, thí dụ sau đây tạo ra một cặp những module rất đơn giản và bạn có thể phối hợp thành một assembly duy nhất. Module đầu tiên là lớp **Fraction**. Lớp này cho phép bạn tạo và thao tác những phân số thông dụng. Thí dụ 3-04 minh họa điều này.

#### *Thí dụ 3-04: Lớp Fraction*

```
*****
using System;

namespace ProgCS
{
    public class Fraction
    {
        public Fraction(int numerator, int denominator)
        {
            this.numerator = numerator;
            this.denominator = denominator;
        }

        public Fraction Add(Fraction rhs)
        {
            if (rhs.denominator != this.denominator)
            {
                throw new ArgumentException("Denominators phải khớp");
            }
            return new Fraction(this.numerator + rhs.numerator,
                                this.denominator);
        }

        public override string ToString()
        {
            return numerator + "/" + denominator;
        }

        private int numerator;
        private int denominator;
    }
}
*****
```

Bạn để ý là lớp **Fraction** nằm trong namespace **ProgCS**. Tên trọn vẹn của lớp là **ProgCS.Fraction**. Lớp **Fraction** nhận hai trị trong hàm constructor: một **numerator** và một **denominator**. Ngoài ra, hàm hành sự **Add()** nhận một **Fraction** thứ hai và trả về tổng cộng với giả định là cả hai cùng chia sẻ một **denominator** chung. Lớp này khá đơn giản nhưng sẽ minh họa chức năng cần thiết cho thí dụ.

Lớp thứ hai mang tên **myCalc**, là một máy tính, được minh họa bởi thí dụ 3-05:

**Thí dụ 3-05: Máy tính.**

```

*****
using System;

namespace ProgCS
{
    public class myCalc
    {
        public int Add(int val1, int val2)
        { return val1 + val2;
        }

        public int Mult (int val1, int val2)
        { return val1 * val2;
        }
    }
}
*****

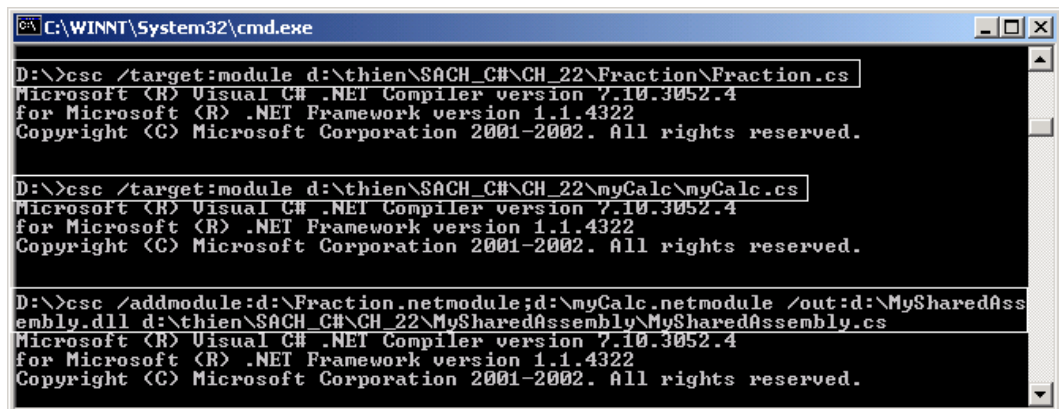
```

Một lần nữa, lớp **myCalc** được viết ra một cách rất đơn giản. Lớp này cũng thuộc namespace **ProgCS**.

Điều này đủ tạo một assembly. Bạn sẽ dùng đến tập tin *AssemblyInfo.cs* để thêm vài metadata vào assembly. Việc sử dụng metadata sẽ được đề cập đến ở chương 5, “Marshalling và Remoting”.

**Bạn để ý:** Bạn có thể tự viết tập tin *AssemblyInfo.cs* nhưng cách tiếp cận đơn giản nhất là để cho Visual Studio .NET kết sinh tự động tập tin này cho bạn.

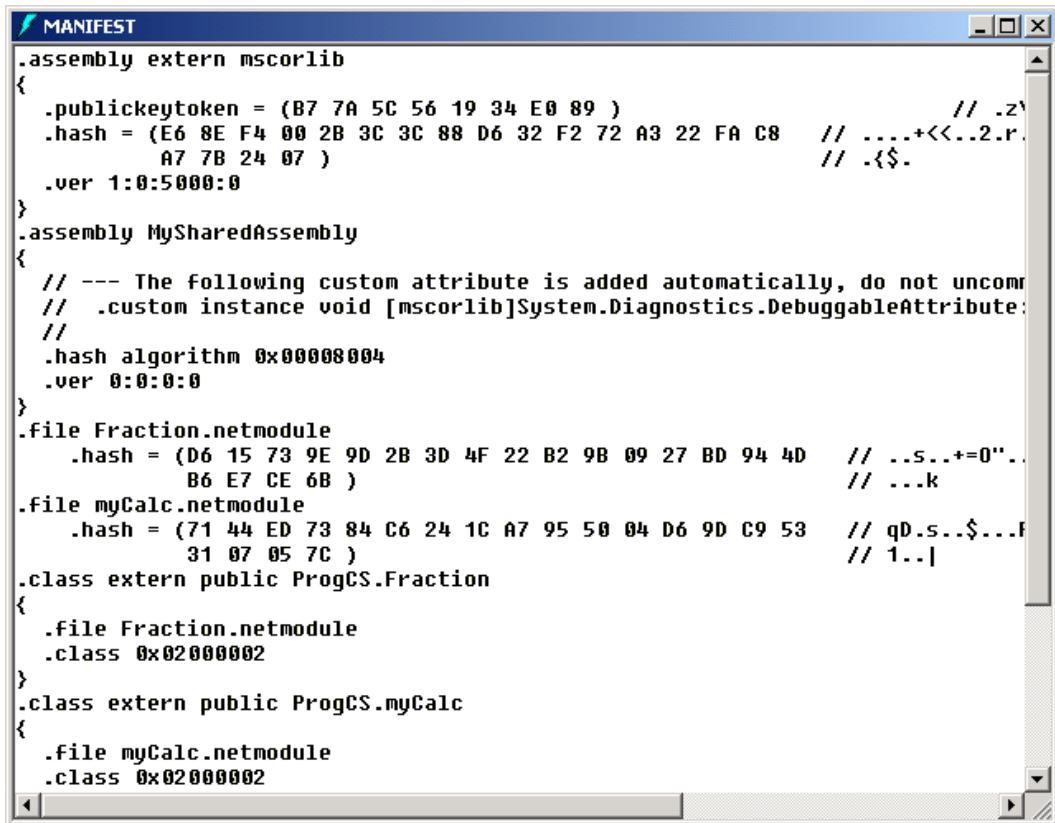
Theo mặc nhiên, Visual Studio .NET sẽ tạo ra single-module assembly. Bạn có thể tạo một multi-module resource với switch **/addModules** của trình tiện ích **csc.exe** trên command line. Hình 3-19 cho thấy việc tạo một multi-module assembly, dùng csc.exe:



Hình 3-19: Dùng csc.exe để tạo multi-module assembly

Trên hình 3-19, bạn thấy có 3 lệnh được đóng khung trong hình chữ nhật. Hai lệnh đầu tiên tạo .netmodule đối với Fraction.dll và myCal.dll. Lệnh sau cùng gom hai .netmodule vào MySharedAssembly.dll. Nếu bạn dùng **ILDasm.exe** cho hiển thị manifest của MySharedAssembly.dll, bạn sẽ thấy hình 3-20 như sau, và bạn thấy là nó chỉ bao gồm một manifest mà thôi.

Và nếu bạn quan sát manifest, thì bạn thấy metadata đối với những library mà bạn đã tạo ra như theo hình 3-20.



Hình 3-20: Manifest của MySharedAssembly.dll

Trước tiên, bạn thấy một assembly nằm ngoài đối với library cốt lõi (**mscorlib**), theo sau bởi hai module **Fraction.netmodule** và **myCalc.netmodule**.

Bây giờ bạn có một assembly gồm 3 tập tin DLL: *fraction.dll*, *calc.dll* (với các kiểu dữ liệu và đoạn mã thi công cần đến) và *MySharedAssembly.dll*. (với manifest).

Ngoài ra, cách dễ nhất để biên dịch và xây dựng một multi-module assembly là với một **makefile**, mà bạn có thể tạo sử dụng Notepad hoặc bất cứ text editor nào.

**Bạn để ý:** Nếu bạn chưa quen với makefile, thì không sao; đây là thí dụ duy nhất sử dụng đến makefile, và đây là cách khắc phục sự hạn chế của Visual Studio .NET khi chỉ tạo những single-module assembly. Nếu thấy cần thiết, bạn có thể chỉ dùng makefile mà chúng tôi cung cấp ở đây mà khỏi phải hiểu tường tận mỗi dòng lệnh.

Thí dụ 3-06 sau đây cho thấy toàn bộ makefile (sẽ được giải thích chi tiết ngay liền sau).

### ***Thí dụ 3-06: Toàn bộ makefile đối với một multi-module assembly***

```
*****
ASSEMBLY = MySharedAssembly.dll

BIN=.\bin
SRC=.
DEST=.\bin

CSC=csc /nologo /debug+ /d:DEBUG /d:TRACE
MODULETARGET=/t:module
LIBTARGET=/t:library
EXETARGET=/:exe

REFERENCES=System.dll

MODULES=$(DEST)\Fraction.dll $(DEST)\Calc.dll
METADATA=$(SRC)\AssemblyInfo.cs

all: $(DEST)\MySharedAssembly.dll

# Assembly metadata được đặt cùng module như là manifest
$(DEST)\$(ASSEMBLY): $(METADATA) $(MODULES) $(DEST)
    $(CSC) $(LIBTARGET) /addmodule:$(MODULES:=;) /out:$@ %s

# Thêm module Calc.dll vào dependency list này
$(DEST)\Calc.dll: Calc.cs $(DEST)
    $(CSC) $(MODULETARGET) /r:$(REFERENCES: =;) /out:$@ %s

# Thêm module Fraction.dll
$(DEST)\Fraction.dll: Fraction.cs $(DEST)
    $(CSC) $(MODULETARGET) /r:$(REFERENCES: =;) /out:$@ %s

$(DEST)::
!if !EXISTS($(DEST))
    mkdir $(DEST)
!endif
*****
```

Makefile bắt đầu bằng cách định nghĩa assembly mà bạn muốn xây dựng:

```
ASSEMBLY = MySharedAssembly.dll
```

Sau đó định nghĩa những thư mục mà bạn sẽ dùng, cho xuất liệu vào thư mục bin nằm dưới thư mục hiện hành và tìm lại mã nguồn từ thư mục hiện hành:

```
BIN=. \bin
SRC=.
DEST=. \bin
```

Bạn xây dựng assembly như sau:

```
$(DEST)\$(ASSEMBLY): $(METADATA) $(MODULES) $(DEST)
    $(CSC) $(LIBTARGET) /addmodule:$(MODULES:=;) /out:$@ %s
```

Lệnh trên sẽ đặt assembly (*MySharedAssembly.dll*) lên thư mục đích (**bin**). Nó bảo **nmake** (chương trình thi hành makefile) là assembly bao gồm metadata và các module, và nó cung cấp command line cần thiết để xây dựng assembly.

Metadata đã được định nghĩa trước đó như sau:

```
METADATA=$(SRC) \AssemblyInfo.cs
```

Các modules được định nghĩa như là hai DLL như sau:

```
MODULES=$(DEST) \Fraction.dll $(DEST) \Calc.dll
```

Dòng lệnh biên dịch xây dựng thư viện và thêm các module, đưa phần kết xuất lên tập tin assembly *MySharedAssembly.dll*:

```
$(DEST)\$(ASSEMBLY): $(METADATA) $(MODULES) $(DEST)
    $(CSC) $(LIBTARGET) /addmodule:$(MODULES:=;) /out:$@ %s
```

Muốn thực hiện việc này, **nmake** cần biết làm thế nào tạo những module. Bạn bắt đầu bảo **nmake** làm thế nào tạo *calc.dll*. Như thế, bạn cần tập tin nguồn *calc.cs*, và bạn bảo **nmake** command line xây dựng DLL này:

```
$(DEST)\Calc.dll: Calc.cs $(DEST)
    $(CSC) $(MODULETARGET) /r:$(REFERENCES:=;) /out:$@ %s
```

Sau đó, bạn cũng làm việc trên đối với *Fraction.dll*:

```
$(DEST)\Fraction.dll: Fraction.cs $(DEST)
    $(CSC) $(MODULETARGET) /r:$(REFERENCES:=;) /out:$@ %s
```

Kết quả việc cho chạy **nmake** đối với makefile này là tạo 3 DLL: *fraction.dll*, *calc.dll* và *MySharedAssembly.dll*. Hình 3-20 cho thấy kết quả.



*Trắc nghiệm assembly*

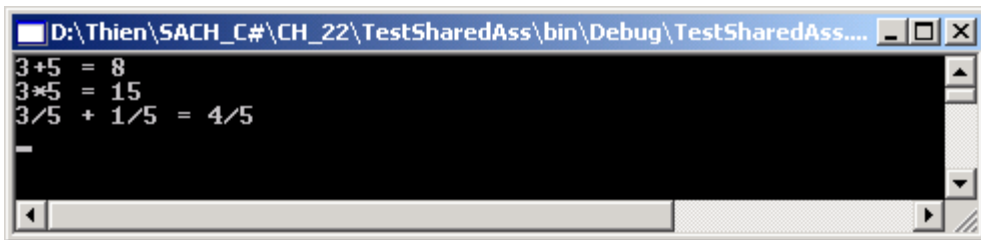
Muốn sử dụng những modules này, bạn cần tạo một chương trình driver sẽ nạp vào những module này khi thấy cần đến. Thí dụ 3-07 sau đây minh họa điều này. Bạn cho cất trữ chương trình dưới cái tên TestSharedAss.cs trong cùng thư mục với các module khác:

**Thí dụ 3-07: Một module test driver**

```
*****
namespace Prog_CSharp
{
    using System;

    public class Test
    {
        // main sẽ không nạp shared assembly
        static void Main()
        {
            Test t = new Test();
            t.UseCS();
            t.UseFraction();
        }
        // triệu gọi này sẽ nạp vào assembly myCalc và mySharedAssembly
        public void UseCS()
        {
            ProgCS.myCalc calc = new ProgCS.myCalc();
            Console.WriteLine("3+5 = {0}\n3*5 = {1}", calc.Add(3,5),
                               calc.Mult(3,5));
        }
        // triệu gọi này sẽ thêm assembly Fraction
        public void UseFraction()
        {
            ProgCS.Fraction frac1 = new ProgCS.Fraction(3,5);
            ProgCS.Fraction frac2 = new ProgCS.Fraction(1,5);
            ProgCS.Fraction frac3 = frac1.Add(frac2);
            Console.WriteLine("{0} + {1} = {2}", frac1, frac2, frac3);
        }
    }
}
*****
```

Hình 3-21 cho thấy kết xuất:



**Hình 3-21: Kết xuất của Test Driver**

Vì mục đích minh họa, điều quan trọng là không được đưa bất cứ đoạn mã nào vào **Main()** lại tùy thuộc vào các module. Bạn không muốn các module sẽ được nạp vào khi **Main()** được nạp, và do đó không có các đối tượng **Fraction** hoặc **Calc** được đưa vào **Main()**. Khi bạn triệu gọi trên **UseFraction** và **UseCalc**, bạn sẽ có khả năng thấy các module được nạp vào riêng rẽ.

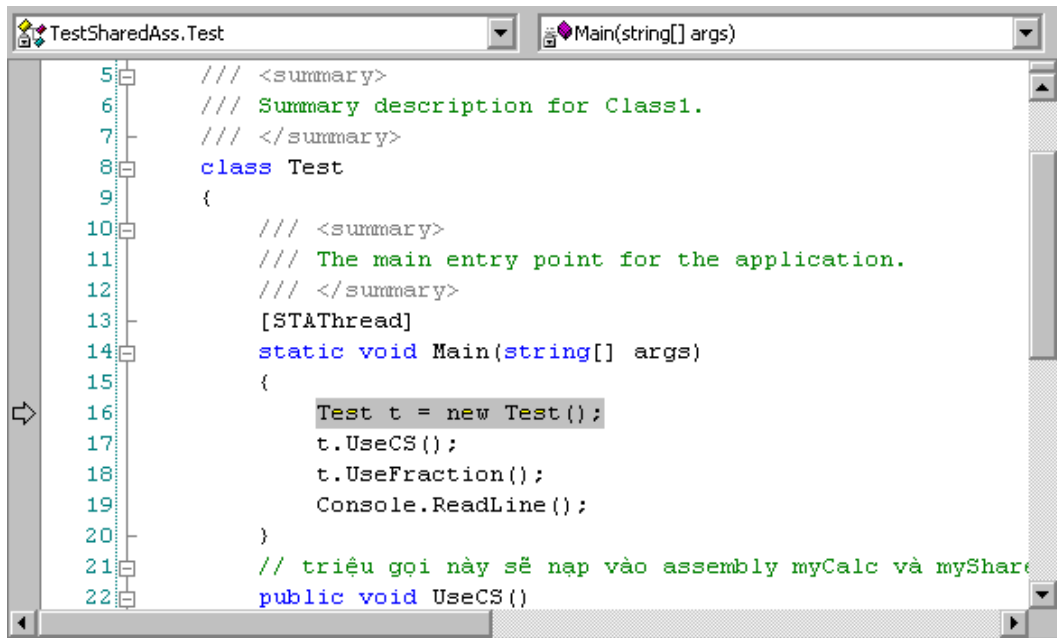
### Nạp assembly vào

Một assembly được nạp vào ứng dụng bởi **AssemblyResolver** thông qua một process được mang tên *probing*. **Assembly resolver** sẽ tự động được triệu gọi bởi .NET Framework; bạn không triệu gọi nó một cách tường minh. Công việc của nó là giải quyết tên assembly đối với một chương trình EXE và nạp chương trình của bạn.

Với một private assembly, **AssemblyResolver** chỉ nhìn vào thư mục nạp của ứng dụng cùng với các thư mục con, nghĩa là thư mục theo đây bạn triệu gọi ứng dụng của bạn.

**Bạn để ý:** 3 DLL được tạo ra trước đó phải nằm cùng thư mục theo đây thí dụ 3-07 được thi hành hoặc trên một thư mục con của thư mục này.

Bạn cho đặt một breakpoint vào hàng thứ hai trên Main, như theo hình 3-22:

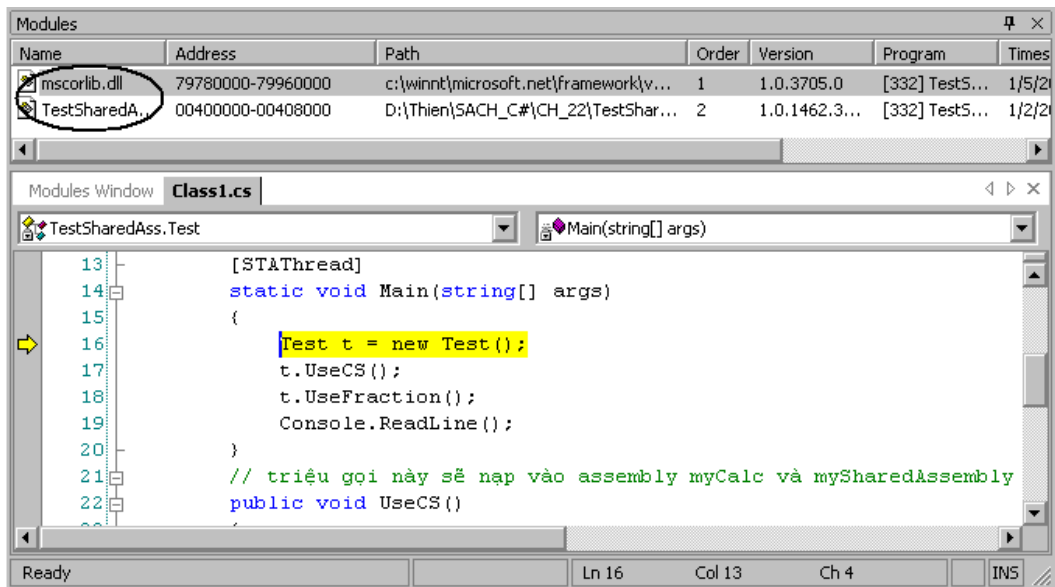


Hình 3-22: Một break point trong hàm Main()

Bạn cho thi hành cho tới break point, rồi cho mở cửa sổ Modules, bằng cách ra lệnh **Debug | Windows | Modules**. Hình 3-23 cho thấy là chỉ hai module **mscorlib.dll** và **TestSharedAss.dll** được nạp vào (nơi chúng tôi cho vòng tròn trên hình 3-23).

Bạn chui vào triệu gọi hàm thứ nhất rồi quan sát cửa sổ Modules. Khi bạn vừa bước vào **UseCalc**, thì **AssemblyLoader** nhận ra nó cần một assembly từ *MyShared Assembly.dll*. DLL được nạp vào, và từ manifest của assembly này, **AssemblyLoader** tìm ra nó cần đến Calc.dll và DLL này được nạp vào như theo hình 3-24.

Khi bạn chui vào **Fraction**, thì DLL cuối cùng, Fraction.dll sẽ được nạp vào. Lợi điểm của multi-module assembly là khi module nào cần đến thì mới được nạp vào, không nạp trước choán chỗ vô ích.

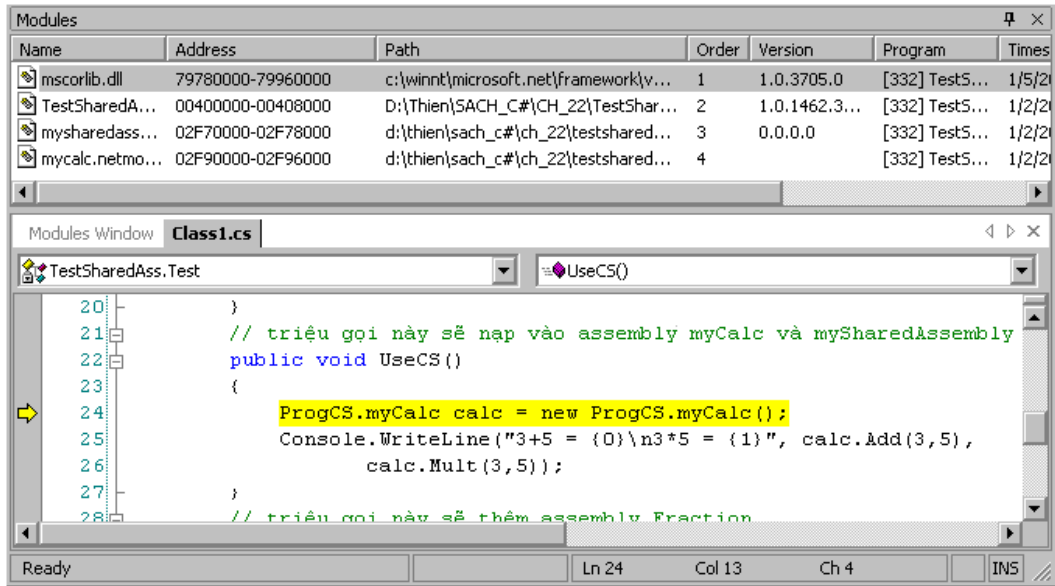


Hình 3-23: Chỉ có hai module được nạp vào

### 3.4 Thử tìm hiểu Private Assembly

Assembly nằm ở hai dạng: *private assembly* và *shared assembly*. Dạng thứ nhất *private assembly* là một collection những kiểu dữ liệu chỉ dành riêng cho một ứng dụng sử dụng, còn dạng thứ hai, *shared assembly* thì lại được chia sẻ sử dụng bởi nhiều ứng dụng. Thí dụ, **CarLibrary.dll** là một private assembly được sử dụng bởi ứng dụng **CSharpCarClient**, và **VBCarClient**. Khi bạn cho đặt đề một qui chiếu về assembly **CarLibrary.dll**, như bạn đã làm trong ứng dụng **CSharpCarClient**, hoặc **VBCarClient** thì IDE sẽ đáp ứng yêu cầu bằng cách sao trộn vẹn assembly cho đặt vào thư mục ứng

dụng của dự án của bạn. Đây là các hành xử mặc nhiên vì private assembly được giả định là mục lựa chọn triển khai.



Hình 3-24: Các module được nạp theo nhu cầu

Tất cả các assembly mà bạn đã xây dựng mãi tới đây đều thuộc loại private assembly. Theo mặc nhiên, khi bạn biên dịch một ứng dụng thì một private assembly sẽ được tạo ra. Các tập tin đối với một private assembly đều nằm trên cùng một thư mục với ứng dụng chủ sở hữu, thường được gọi là *application directory* (hoặc trên một cây thư mục con, được cách ly khỏi phần còn lại của hệ thống), và không được chia sẻ sử dụng bởi các ứng dụng khác. Cây thư mục con hoạt động giống như ứng dụng tùy thuộc vào nó và bạn có thể triển khai ứng dụng này trên một máy khác bằng cách chép thư mục và thư mục con kèm theo.

Một private assembly có thể có bất cứ tên nào bạn chọn đặt cho. Không hề gì nếu tên này đụng độ với những assembly trên các ứng dụng khác; tên sẽ mang tính cục bộ chỉ đối với một ứng dụng duy nhất.

Trong quá khứ, DLL thường được cài đặt trên một máy và điểm đột nhập vào DLL được ghi nhận trong **Windows Registry**. Rất khó lòng tránh khỏi việc Registry bị phá bình. và cài đặt chương trình trên một máy khác không phải là điều đơn giản. Với assembly, những điều phiền toái này sẽ biến mất. Với private assembly, việc cài đặt rất đơn giản giống như là sao chép một tập tin vào một thư mục thích ứng. Điều quan trọng là bạn khỏi bận tâm khi việc gỡ bỏ một private assembly sẽ gây trục trặc đối với bất cứ ứng dụng nào đó trên máy.

Việc giải quyết tìm ra nơi tá túc của assembly cũng như việc nạp private assembly **CarLibrary.dll** được thực hiện dựa theo sự kiện là assembly được đưa vào thư mục ứng dụng. Kỳ thật, nếu bạn di chuyển ứng dụng **CSharpCarClient.exe** và **CarLibrary.dll** về một thư mục mới, thì ứng dụng vẫn chạy như thường.

Việc gỡ bỏ cài đặt (uninstalling) hoặc sao y (replicating) một ứng dụng đối với private assembly cũng diễn ra không rắc rối. Bạn chỉ cần **Delete** (hoặc **Copy**) thư mục ứng dụng, và bạn có thể yên tâm mà ngủ vì việc làm này sẽ không phá bĩnh những ứng dụng khác trên máy.

### 3.4.1 Cơ bản về Probing

.NET Runtime giải quyết việc tìm ra nơi tá túc của một private assembly bằng cách sử dụng một kỹ thuật được gọi là *probing*<sup>10</sup>. Đây là một tiến trình ánh xạ một qui chiếu assembly nằm ngoài (external assembly reference), nghĩa là [.assembly extern], về đúng tập tin nhị phân tương ứng. Thí dụ, khi .NET Runtime đọc hàng sau đây từ một manifest:

```
.assembly extern CarLibrary
{
  . . .
}
```

thì một cuộc dò tìm sẽ được thực hiện trên thư mục ứng dụng đối với một tập tin mang tên **CarLibrary.dll**. Nếu không thể tìm thấy nơi tá túc của một DLL binary, thì lại thử đối với một **CarLibrary.exe**. Bạn nhớ cho là nếu assembly chứa tag [.publickeytoken], nghĩa là liên quan đến một shared assembly, thì GAC sẽ được dò tìm trước tiên (ta sẽ xem chi tiết sau trong chốc lát).

#### 3.4.1.1 Nhận diện một Private Assembly

Một private assembly sẽ được nhận diện thông qua tên dạng chuỗi và một con số phiên bản, cả hai được ghi nhận trên assembly manifest. Cái tên sẽ được tạo ra dựa trên tên của binary module chứa assembly manifest. Thí dụ, nếu bạn quan sát manifest của assembly **CarLibrary.dll**, bạn sẽ tìm thấy như sau (phiên bản của bạn có thể khác đi):

```
.assembly CarLibrary
{
  . . .
  .ver 1:0:454:30142
}
```

Tuy nhiên, xét theo bản chất của một private assembly, thì không có gì ngạc nhiên

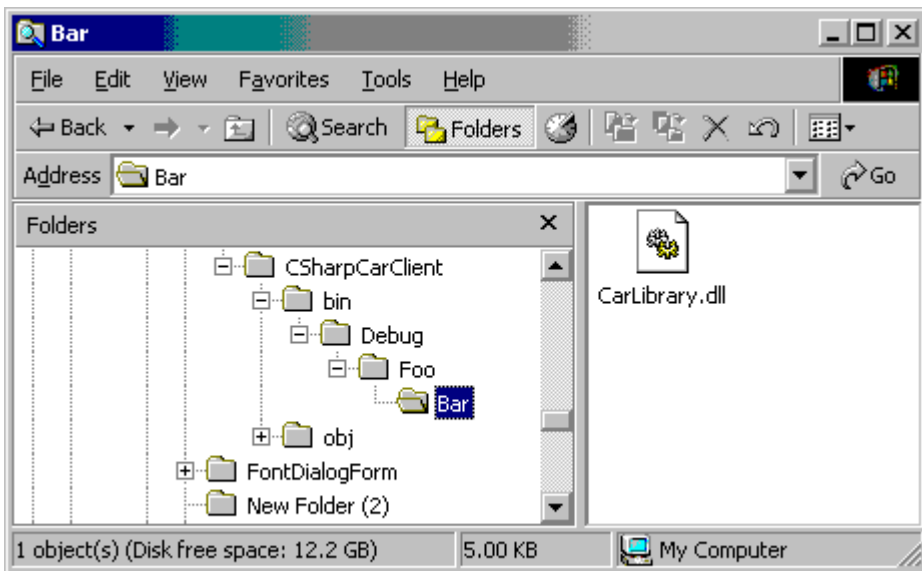
---

<sup>10</sup> Probing là thăm dò, điều tra.

khi CLR sẽ không bận tâm áp dụng bất cứ cơ chế kiểm tra phiên bản nào khi nạp assembly. Người ta giả định là *không cần kiểm tra về phiên bản đối với private assembly* vì rằng ứng dụng client là thực thể duy nhất “biết được” sự hiện diện của nó. Ngoài ra, bạn cũng nên nhớ là có thể đối với một máy đơn lẻ ta sẽ có nhiều bản sao của cùng private assembly trên nhiều thư mục ứng dụng khác nhau.

### 3.4.1.2 Private Assembly và các tập tin XML Configuration

Khi .NET Runtime nhận được chỉ thị gắn kết với một assembly, điều đầu tiên là xác định sự hiện diện của một tập tin cấu hình ứng dụng (application configuration file). Tập tin tùy chọn này chứa những tag XML điều khiển cách gắn kết (binding) của ứng dụng khởi động (launching application). Theo nguyên tắc, các tập tin cấu hình phải mang cùng tên như ứng dụng khởi động kèm theo cái đuôi \*.config.



Hình 3-25: Đổi chỗ tá túc đối với assembly của Bạn

Các tập tin .config có thể được dùng khai báo bất cứ thư mục con nào mà ta có thể lục tìm trong suốt quá trình gắn kết với private assembly. Như ta đã nói, một ứng dụng .NET có thể được triển khai bằng cách đưa tất cả các assembly ghi lên cùng thư mục với ứng dụng khởi động. Tuy nhiên, thường xuyên, có thể bạn mong muốn triển khai một ứng dụng mà theo đây thư mục ứng dụng sẽ chứa một số thư mục con liên đới để có thể đem lại một cấu trúc có ý nghĩa đối với ứng dụng như là một tổng thể.

Bao giờ bạn cũng thấy điều này trong các phần mềm thị trường. Thí dụ, giả sử thư mục chính của ta mang tên MyRadApp, chứa một số thư mục con (\Images, \Bin,

\SavedGames, v.v..). Bằng cách sử dụng một tập tin .config bạn có thể ra lệnh cho CLR phải dò tìm ở đâu khi muốn tìm ra nơi tá túc của các assembly được dùng bởi ứng dụng khởi động.

Để minh họa, ta thử tạo một tập tin \*.config đơn giản đối với ứng dụng **CSharpCarClient**. Mục đích của chúng ta là di lý assembly được qui chiếu (CarLibrary.dll) từ thư mục **Debug** vào một thư mục con mới mang tên **Foo\Bar**. Bây giờ, bạn tiến hành như theo hình 3-25, trang trước.

Bây giờ bạn tạo một tập tin \*.config mang tên **CSharpCarClient.exe.config** (dùng Notepad cũng đủ) và cho cất trữ lên cùng thư mục chứa ứng dụng CSharpCarClient.exe. Đầu tập tin cấu hình của một ứng dụng được đánh dấu bởi tag <Configuration>. Trước tag khép </Configuration>, ta khai báo một hàng **assemblyBinding**, được sử dụng khai báo một nơi tá túc thay thế (alternative) để truy tìm đối với một assembly nào đó, sử dụng attribute **privatePath**. (nếu nhiều thư mục thì sử dụng dấu chấm phẩy để ngăn cách danh sách):

```
<configuration>
  <runtime>
    <assemblyBinding xmlns="urn:schemas-microsoft-com:asm.v1">
      <probing privatePath="foo\bar"/>
    </assemblyBinding>
  </runtime>
</configuration>
```

Một khi bạn khở xong nội dung tập tin cấu hình, cho cất trữ tập tin rồi khởi động CsharpCarClient, bạn sẽ thấy là ứng dụng chạy suôn sẻ.

Như là trắc nghiệm cuối cùng, bạn cho đổi tên tập tin \*.config (chẳng hạn NoCSharpCarClient.exe.config) rồi cho chạy lại lần nữa. Lúc này ứng dụng client im re, coi như thất bại. Bạn nhớ lại là tập tin cấu hình phải mang cùng tên với ứng dụng khởi động. Bởi vì bạn cho đổi tên tập tin .config nên .NET Runtime xem như bạn không có một tập tin cấu hình, nên do đó đi dò tìm assembly được qui chiếu trong thư mục ứng dụng (nhưng không tìm thấy).

### 3.4.1.3 Những đặc thù trong việc gắn kết với một private assembly

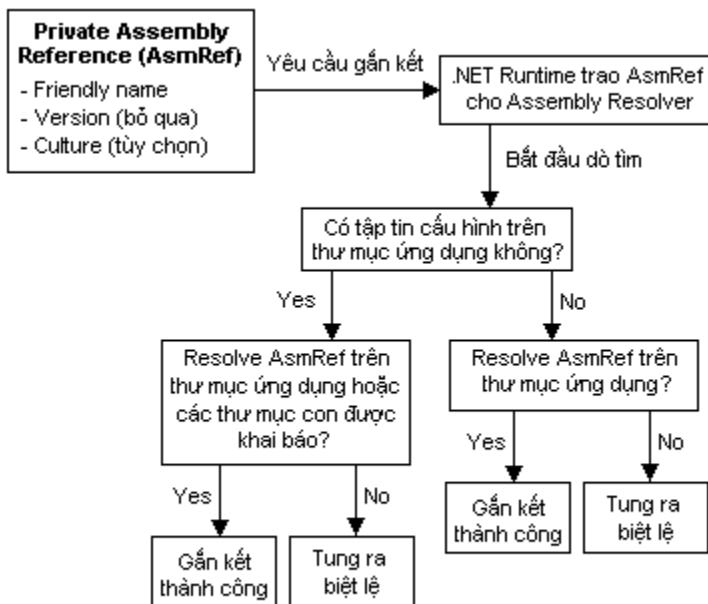
Để đóng khung lại việc bàn cãi này, ta thử tóm lược lại các bước cụ thể trong việc gắn kết với một private assembly vào lúc chạy. Trước tiên, một yêu cầu nạp một assembly có thể hoặc *explicit* hoặc *implicit*. Một yêu cầu nạp ngầm hiệu (implicit) xảy ra bất cứ lúc nào khi manifest đưa ra một qui chiếu trực tiếp về assembly ngoại lai nào đó. Như bạn còn nhớ, external reference được đánh dấu bởi tag [.assembly extern]:

```
// Một yêu cầu nạp implicit
.assembly extern CarLibrary
{
    . . .
}
```

Một yêu cầu nạp tường minh (explicit) xảy ra bằng chương trình sử dụng hàm hành sự **System.Reflection.Assembly.Load()**. Hàm hành sự **Load()** cho phép bạn khai báo tên, phiên bản, strong name, và thông tin culture:

```
// Một yêu cầu explicit
Assembly asm = Assembly.Load("CarLibrary");
```

Một cách tổng thể, tên, phiên bản, strong name và thông tin culture được gọi là một *assembly reference* (gọi tắt AsmRef). Bộ phận lo dò tìm nơi tá túc của đúng assembly dựa trên một AsmRef được gọi là *assembly resolver*, được xem như là một tiện ích của CLR.



**Hình 3-26: Dò tìm đối với một private assembly**

.NET Runtime chuyển qua cho assembly resolver một AsmRef. Nếu bộ resolver xác định là AsmRef qui chiếu về một private assembly (nghĩa là không có strong name được ghi trong manifest), thì các bước sau đây sẽ được thực hiện (hình 3-26):

Như đã nói, một thư mục ứng dụng chỉ đơn giản là một thư mục trên đĩa cứng (thí dụ, C:\MyApp) chứa tất cả các tập tin đối với một ứng dụng nào đó. Nếu thấy cần thiết, một thư mục ứng dụng có thể khai báo những thư mục con bổ sung (chẳng hạn C:\MyApp\Bin, C:\MyApp\Tools, v.v..) để thiết lập một cây đẳng cấp các tập tin.

Khi một yêu cầu gắn kết được phát ra,



1. Trước tiên, assembly resolver cố gắng tìm ra một tập tin cấu hình trên thư mục ứng dụng. Như bạn có thể thấy, tập tin này cho biết những thư mục con bổ sung cần được đưa vào trong truy tìm, cũng như thiết lập một chính sách về phiên bản được sử dụng đến đối với việc gắn kết hiện hành.
2. Nếu không có tập tin cấu hình, .NET Runtime sẽ cố gắng phát hiện đúng assembly bằng cách xem xét thư mục ứng dụng hiện hành. Nếu tập tin cấu hình hiện hữu bất cứ những thư mục con được khai báo sẽ được truy tìm.
3. Nếu assembly không tìm thấy trong lòng thư mục ứng dụng (hoặc trên thư mục con được khai báo) thì việc truy tìm ngưng ở đây, và một biệt lệ kiểu `TypeLoadException` sẽ được tung ra.

Hình 3-26 minh họa tiến trình dò tìm assembly vừa kể trên.

Một lần nữa, bạn có thể thấy nơi tá túc của một private assembly có thể được tìm thấy một cách khá đơn giản. Nếu thư mục ứng dụng không chứa một tập tin cấu hình, thì assembly resolver sẽ đơn giản giải quyết bằng cách nhìn xem một binary nào khớp đúng với chuỗi tên. Nếu thư mục ứng dụng không chứa tập tin cấu hình, thì bất cứ thư mục con nào được khai báo cũng sẽ được truy tìm tiếp.

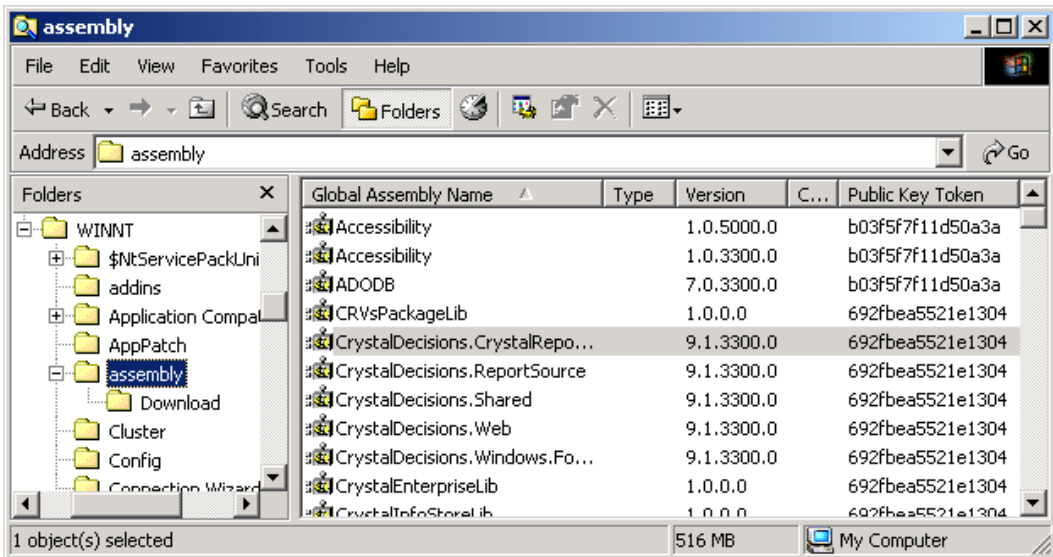
## 3.5 Thử tìm hiểu Shared Assembly

Bạn có thể tạo ra những assembly được chia sẻ sử dụng bởi những ứng dụng khác. Nếu bạn viết ra một ô control mang tính chung chung (generic control) hoặc một lớp có thể được chia sẻ sử dụng bởi nhiều nhà triển khai khác, đó là bạn đang thực hiện một shared assembly. Shared assembly có thể được sử dụng bởi nhiều khách hàng trên một máy duy nhất. Rõ ràng là nếu bạn muốn tạo một thư viện lớp mang tính phổ biến lên toàn bộ máy tính (machine-wide) thì nên thực hiện những shared assembly. Nếu bạn muốn chia sẻ sử dụng assembly bạn viết ra, bạn phải theo một số yêu cầu khắt khe.

Trước tiên, assembly của bạn phải có một *strong name*, một tên duy nhất được chia sẻ sử dụng. Thứ đến, shared assembly của bạn phải được bảo vệ không cho các phiên bản mới phá bình, như vậy phải có cách kiểm tra phiên bản trước khi được nạp vào theo yêu cầu của ứng dụng client. Cuối cùng, muốn cho chia sẻ sử dụng assembly bạn tạo ra, bạn phải đưa assembly này vào *Global Assembly Cache* (GAC). Đây là một vùng ký ức của hệ thống các tập tin được dành riêng bởi CLR để trữ các shared assembly. Bản thân GAC nằm ở thư mục <ổ đĩa>:\WINNT\Assembly. Trên .NET, các shared assembly đều được tập trung đưa vào một nơi là GAC. Xem hình 3-27.

### 3.5.1 Kết thúc cơn ác mộng DLL

Với việc sử dụng assembly, cơn ác mộng DLL mà bạn gặp phải trong Windows về trước sẽ chấm dứt. Nếu bạn đã kinh qua lập trình Windows, chắc bạn còn nhớ kịch bản này: bạn cho cài đặt ứng dụng A trên máy của bạn, và nó nạp một số DLL vào trong thư mục Windows. Nó hoạt động ngon lành trong nhiều tháng năm. Sau đó bạn cho cài đặt ứng dụng B cũng trên máy của bạn và thành tình không chờ đợi, ứng dụng A “sụm bả chề không nói không rằng”. Mà ứng dụng B thì lại không dính dáng chỉ với ứng dụng A. Việc gì đã xảy ra? Cuối cùng bạn phát hiện ra, là ứng dụng B đã thay thế một DLL mà ứng dụng A cần đến, và thành tình ứng dụng A bắt đầu lảo đảo và bất động.



Hình 3-27: Global Assembly Cache (GAC)

Khi DLL được sáng tạo, chỗ trữ trên đĩa rất quý, nên ý niệm tái sử dụng DLL là điều tốt lành. Theo nguyên tắc, DLL phải được tương thích lui về sau (backward-compatible), do đó việc tự động nâng cấp lên một DLL mới không khó khăn và an toàn.

Khi DLL mới được thêm vào máy, thì ứng dụng cũ xưa thành tình được kết nối với một DLL không tương thích với những gì ứng dụng chờ đợi và thế là xong đời. Hiện tượng này là một trong những lý do khách hàng sử dụng Windows phải cài đặt ngoài ý muốn phần mềm mới hoặc nâng cấp các chương trình hiện hữu và cũng là một trong những lý do bảo rằng các máy sử dụng Windows là bất ổn. Với assembly thì cơn ác mộng sẽ biến mất.

## 3.5.2 Tìm hiểu cơ chế về phiên bản

Như bạn đã biết, .NET Runtime không buồn kiểm tra phiên bản đối với các private assembly. Nhưng khi “chơi” với shared assembly thì vấn đề phiên bản lại trở thành quan trọng. Bây giờ, ta thử tìm hiểu cơ chế đánh số phiên bản (gọi là versioning) và kiểm tra đối với shared assembly.

Trên .NET, các shared assembly sẽ được nhận diện là duy nhất thông qua tên và phiên bản. GAC cho phép những phiên bản khác “sống chung cùng nhà, tay trong tay” (thường được gọi là side-by-side) nghĩa là phiên bản cũ vẫn hiện diện cùng với phiên bản mới hơn. Như vậy, một ứng dụng đặc biệt nào đó có thể bảo “cho tớ phiên bản mới toanh nhất” hoặc “cho tớ build chốt nhất của Version 2”, hoặc kể cả “chỉ cho tớ phiên bản mà ứng dụng đang sử dụng”.

Một con số đánh dấu phiên bản của một assembly trông giống như sau: 1:0:2204:21, nghĩa là dãy 4 con số phân cách bởi dấu hai chấm. Hai số đầu (1:0) là phiên bản **major** và **minor**. Con số thứ ba (2204) là số **build**, còn con số thứ tư (21) là số **revision** (số chỉnh sửa duyệt lại). Xem hình 3-28.

Major	Minor	Build	Revision
1	0	2204	21
Incompatible		Có thể compatible	QFE

**Hình 3-28: Cấu trúc số phiên bản**

Khi hai assembly có những số major hoặc minor khác nhau (chẳng hạn 1.0 so với 1.5) thì được xem như là *hoàn toàn bất tương thích* (incompatible). Khi các assembly khác nhau khá xa dựa trên con số Major và Minor, thì bạn có thể chắc chắn là có nhiều thay đổi đáng kể (nghĩa là thay đổi tên các hàm hành sự, kiểu dữ liệu được thêm vào hoặc bị gỡ bỏ, các thông số bị thay đổi, v.v.). Do đó, nếu ứng

dụng client yêu cầu gắn kết với phiên bản 2.0 nhưng GAC chỉ chứa phiên bản 2.5 thì yêu cầu gắn kết thất bại (trừ phi bị phủ quyết bởi tập tin cấu hình ứng dụng).

Khi hai phiên bản mang cùng số major và minor nhưng lại có số build khác nhau (chẳng hạn 2.5.0.0 so với 2.5.1.0) thì .NET Runtime giả định chúng *có thể là tương thích* với nhau (nói cách khác xem như có tương thích lui nhưng không bảo đảm). Cuối cùng, khi ba con số đầu giống nhau, chỉ khác số revision (còn gọi là Quick Fix Engineering, QFE) thì được xem như là hoàn toàn tương thích.

Số duyệt lại QFE dành cho những lần sửa chữa bug. Nếu bạn sửa một bug và cho biết là DLL của bạn hoàn toàn tương thích đối với phiên bản hiện hữu, thì bạn phải tăng con số duyệt lại. Khi một ứng dụng nạp một assembly, thì nó cho biết phiên bản major và minor nó muốn nạp, và AssemblyResolver sẽ tìm ra con số build và revision cao nhất.

### 3.5.3 Tìm hiểu về Strong Name

Để có thể sử dụng một shared assembly, bạn cần tuân thủ 3 đòi hỏi sau đây:

- Bạn phải có khả năng khai báo đúng assembly mà bạn muốn nạp vào. Do đó, bạn cần có một tên duy nhất mang tính toàn cục (global unique name) được gán cho shared assembly.
- Bạn cần bảo đảm là assembly không bị giả mạo. Nghĩa là, bạn cần có một digital signature (dấu ấn kỹ thuật số) khi assembly được tạo dựng.
- Bạn cần bảo đảm là assembly bạn đang nạp vào đúng là assembly được phép của tác giả tạo ra assembly. Do đó, bạn cần ghi nhận mã nhận diện người sáng tác (originator).

Do đó, khi bạn muốn tạo ra một assembly có thể được chia sẻ sử dụng bởi nhiều ứng dụng khác nhau trên một máy tính nào đó, bước đầu tiên là tạo ra một tên chia sẻ duy nhất đối với assembly. Tên này được gọi là strong name thường chứa những thông tin sau đây:

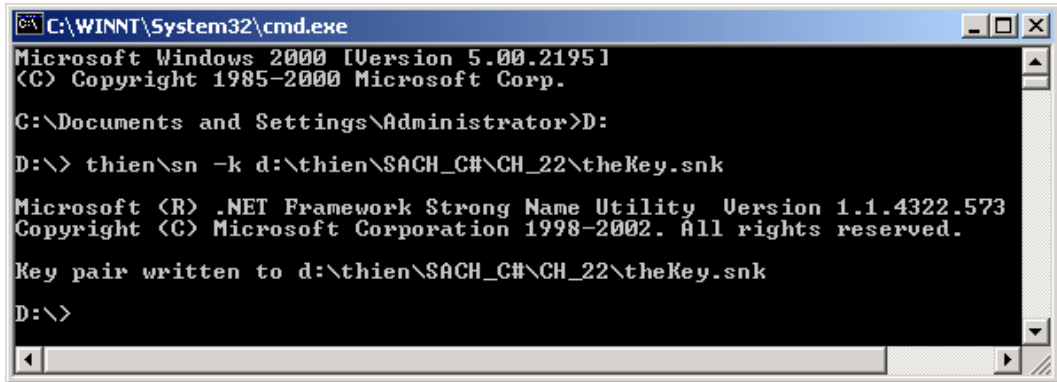
- Một tên kiểu string mang tính thân thiện và thông tin tùy chọn về culture (giống như với private assembly)
- Một mã nhận diện phiên bản
- Một cặp mục khóa public-private
- Một dấu ấn kỹ thuật số (digital signature)

Việc hình thành một strong name đều được dựa trên một phương thức mật mã hóa chuẩn mục khóa công cộng (standard public key encryption). Khi bạn tạo một shared assembly bạn phải kết sinh một cặp mục khóa public-private. Một đoạn mã băm (hash code) được lấy ra từ những tên và nội dung của các tập tin trong assembly. Mã băm này sẽ được mật mã hóa với một private key đối với assembly và được đưa vào manifest. Đây được biết là ký tên assembly (*signing the assembly*). Public key sẽ được sát nhập vào strong name của assembly. Nói tóm lại, một cặp mục khóa sẽ được bao gồm trong chu kỳ build sử dụng một trình biên dịch .NET, trình này sẽ liệt kê một token của public key lên manifest của assembly (thông qua tag `[.publickeytoken]`). Private key sẽ không được liệt kê ra trên manifest nhưng lại được “ký tên” với public key. Dấu ấn kết quả sẽ được trữ ngay trong assembly (trong trường hợp multifile assembly, thì public key sẽ được trữ trên tập tin định nghĩa assembly manifest).

Khi một ứng dụng cho nạp assembly vào, thì CLR sẽ dùng public key để giải mã đoạn mã băm đối với các tập tin thuộc assembly bảo đảm là chúng không bị giả mạo. Việc này cũng bảo vệ chống đụng độ về tên.

### 3.5.4 Xây dựng một shared assembly

Muốn kết sinh một strong name bạn sử dụng trình tiện ích **sn.exe**. (nằm trên thư mục C:\Program Files\Microsoft.NET\SDK\v1.1\Bin).



Hình 3-29: Tạo một tập tin \*.snk

Mặc dù trình tiện ích này có nhiều mục chọn, ta chỉ cần dùng flag “-k” yêu cầu trình tiện ích kết sinh một strong name mới và được trữ trên một tập tin được khai báo (hình 3-29). Thí dụ:

```
sn -k d:\thien\SACH_C#\CH_22\theKey.snk
```

Bạn có thể đặt tên tập tin bất cứ tên gì bạn muốn. Bạn nhớ cho strong name là một chuỗi hexadecimal digit và không dành cho người đọc. Nếu bạn quan sát nội dung của tập tin mới này (theKey.snk), bạn thấy binary markings của cặp key. Xem hình 3-30.

Để tiếp tục với thí dụ, giả sử bạn tạo ra một C# Class Library mới mang tên **SharedAssembly** chẳng hạn. Lớp này chứa định nghĩa sau đây, thí dụ 3-08:

#### Thí dụ 3-08: Lớp SharedAssembly

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

namespace SharedAssembly
{
    public class VWMiniVan
    {
        public VWMiniVan() {}
    }
}
```

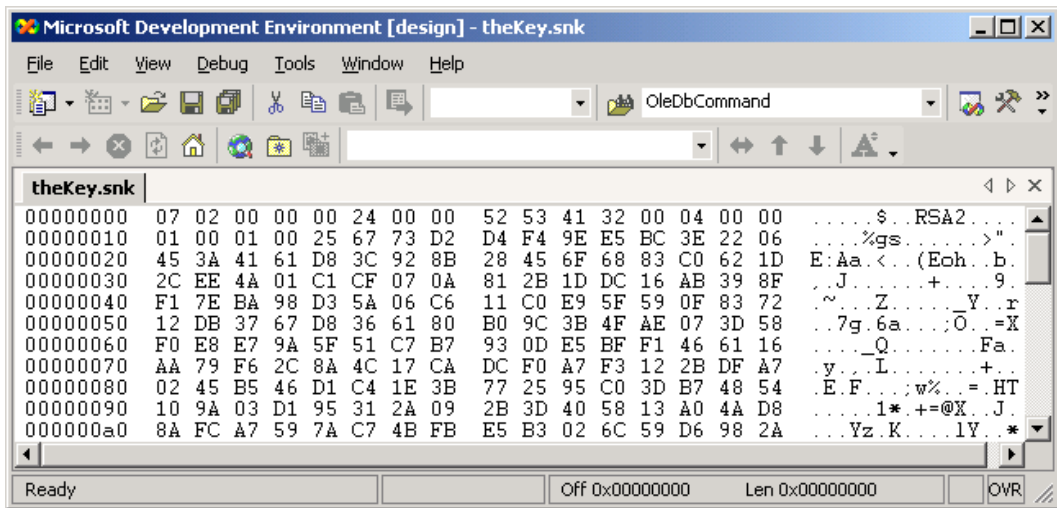
```

public void Play60sTunes()
{
    MessageBox.Show("What a loooong, strange trip it's been...");
}

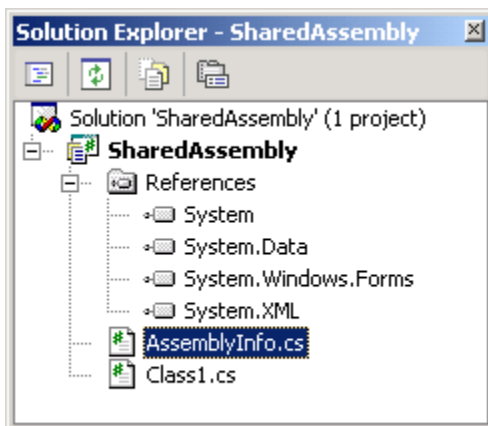
private bool isBustedByTheFuzz = false;

public bool Busted
{
    get { return isBustedByTheFuzz; }
    set { isBustedByTheFuzz = value; }
}
}
}
*****

```



Hình 3-30: Nội dung tập tin theKey.snk (dùng Visual Studio .NET 2002 để đọc hiển thị)



Hình 3-31: Tập tin AssemblyInfo.cs

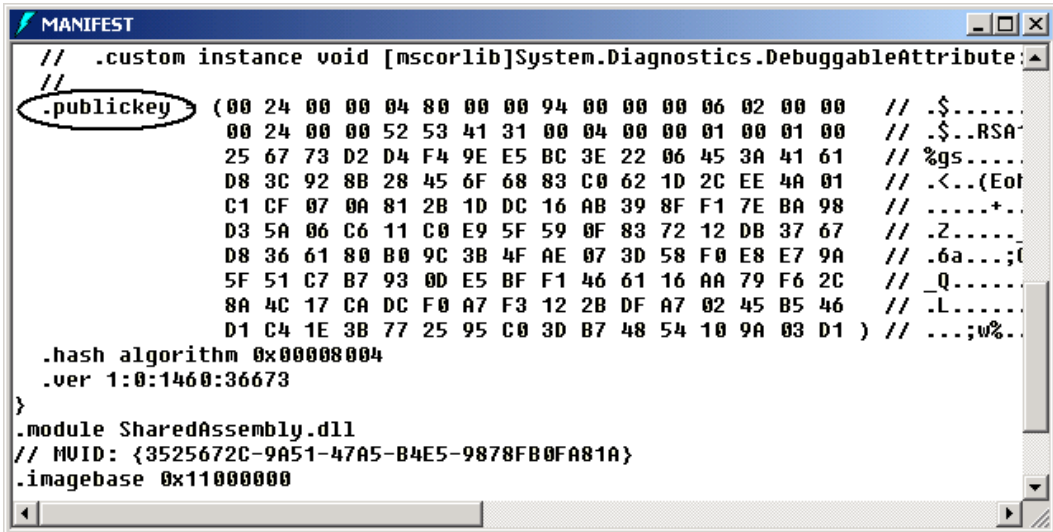
Muốn khai báo một strong name đối với một shared assembly, bạn chỉ cần nhậ tu trị của attribute này cho biết nơi tá túc của tập tin \*.snk.

Bước kế tiếp là cho ghi nhận public key lên assembly manifest. Cách dễ nhất là sử dụng attribute **AssemblyKeyFile**. Khi bạn tạo một C# project workspace, có thể bạn để ý một trong những tập tin khởi sự của dự án mang tên "AssemblyInfo.cs". Xem hình 3-31.

Tập tin này chứa một số attributes (khởi đi là trống rỗng) được tiêu thụ bởi một trình biên dịch .NET. Nếu bạn quan sát tập tin này, bạn sẽ thấy attribute **AssemblyKeyFile** này.

```
[assembly:AssemblyKeyFile(@"D:\Thien
\SACH_C#\CH_22\theKey.snk")]
```

Sử dụng assembly level attribute này, C# compiler giờ đây trộn với nhau thông tin cần thiết vào manifest tương ứng. Bạn có thể dùng ILDasm.exe để xem tag [.publickey]. Xem hình 3-32.



Hình 3-32: Đánh dấu của một shared assembly

### 3.5.5 Cài đặt assembly vào GAC

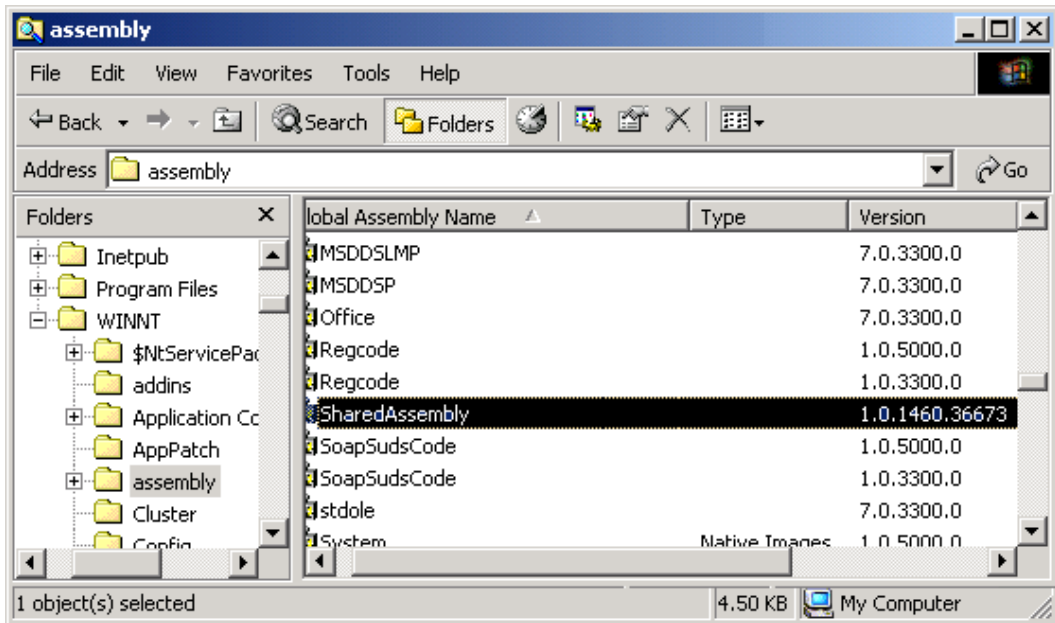
Một khi bạn đã thiết lập một strong name đối với một shared assembly, bước cuối cùng là cho cài đặt nó vào GAC, một thư mục hệ thống được dành riêng. Cách đơn giản nhất để cài đặt một shared assembly vào GAC là cho mở Window Explorer rồi lôi thả tập tin **SharedAssembly.dll** vào thư mục GAC hiện dịch (active window). Muốn thấy GAC, bạn cho mở Windows Explorer rồi hướng về \WINNT\Assembly; File Explorer sẽ nhảy qua một trình tiện ích GAC. Xem hình 3-33. Bạn kiểm tra số phiên bản trên hình 3-32 và 3-33.

Hoặc bạn cũng có thể tự do dùng trình tiện ích gacutil.exenhiện tiện ích trên command line như sau:

```
gacutil /i: SharedAssembly.dll
```

Bạn nên nhớ là bạn phải có quyền Administrative đối với máy tính để có thể cài đặt assembly lên GAC. Điều này tốt thôi, vì nó ngăn ngừa người sử dụng nào đó phá bĩnh một cách vô tình những ứng dụng hiện hữu.

Kết quả cuối cùng là assembly của bạn giờ đây nằm trong GAC và có thể được chia sẻ sử dụng bởi nhiều ứng dụng trên máy tính đích. Bạn để ý, khi nào bạn muốn gỡ bỏ một assembly khỏi GAC, bạn chỉ cần right-click lên assembly, rồi chọn click mục Delete trên trình đơn shortcut.



Hình 3-33: Cài đặt SharedAssembly.dll vào GAC

### 3.5.6 Sử dụng một Shared Assembly

Bây giờ để chứng minh, giả sử bạn tạo một ứng dụng C# Console Application mới, cho mang tên **SharedAssemblyUser** chẳng hạn, cho qui chiếu về SharedAssembly binary, và bạn định nghĩa lớp sau đây, thí dụ 3-09:

#### Thí dụ 3-09: Lớp SharedAssemblyUser

```

*****
namespace SharedLibUser
{
    using System;
    using SharedAssembly;

    public class SharedAsmUser
    {
        public static int Main(string[] args)
    }
}

```





Nếu bạn cho gỡ bỏ SharedAssembly.dll trên GAC, rồi cho chạy lại chương trình thì một biệt lệ System.IO.FileNotFoundException sẽ được tung ra.

### 3.5.7 Ghi nhận thông tin về Version

Tới đây, có thể bạn hỏi con số phiên bản được khai báo ở đâu? Bạn nhớ lại là mỗi dự án C# đều định nghĩa một tập tin mạng tên **AssemblyInfo.cs**. Nếu bạn quan sát tập tin này (bạn mở dự án **CarLibrary.dll** chẳng hạn, gọi *Solution Explorer* vào rồi double-click lên **AssemblyInfo.cs** thì cửa sổ tập tin này sẽ hiện lên). Bạn thấy một chuỗi như sau được đặt để:

```
[assembly: AssemblyVersion("1.0.*")]
```

Mỗi dự án mới C# bắt đầu cuộc sống với phiên bản 1.0. Khi bạn build phiên bản mới của một shared assembly, phần lớn công việc của bạn là nhật tu con số thứ tư của phiên bản. Bạn nên nhớ là Visual Studio .NET IDE sẽ tự động tăng con số build và revision (được đánh dấu bởi tag “\*”). Bạn có thể nhật tu chuỗi trên như sau:

```
[assembly: AssemblyVersion("1.0.0.0")]
```

#### 3.5.7.1 “Đông cứng” SharedAssembly hiện hành

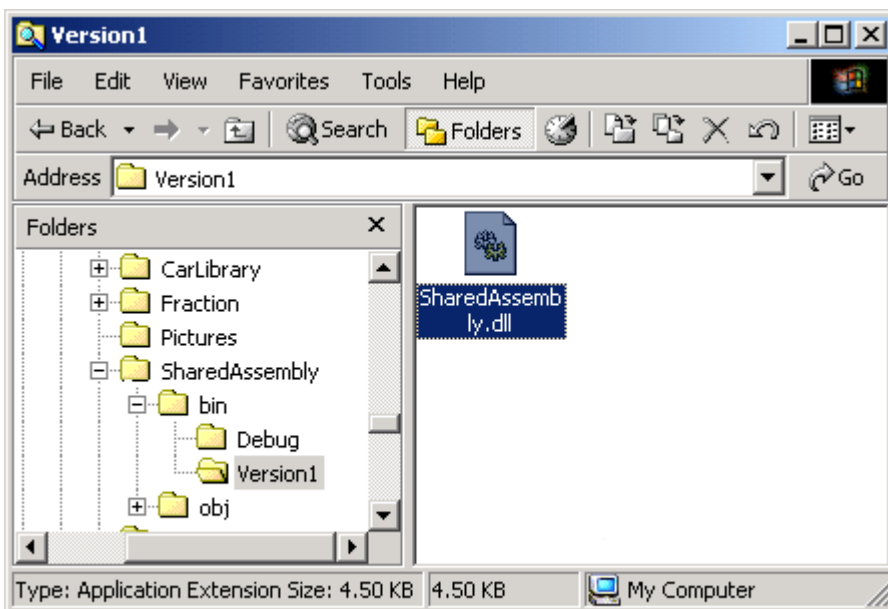
Muốn thật sự hiểu thấu cơ chế phiên bản trên .NET, ta cần một thí dụ cụ thể. Mục tiêu hiện hành là cho nhật tu assembly trước đây, **SharedAssembly.dll** cho phép hỗ trợ những chức năng mới, cho nhật tu số phiên bản, rồi sau đó đưa phiên bản mới vào GAC. Tới thời điểm này, bạn có khả năng thử nghiệm thông qua tập tin \*.config khai báo những cơ chế phiên bản khác nhau, cũng như việc thi hành “tay trong tay” (side-by-side).

Để bắt đầu, ta thử nhật tu hàm constructor của lớp VWMiniVan cho phép hiển thị một thông điệp kiểm tra phiên bản hiện hành:

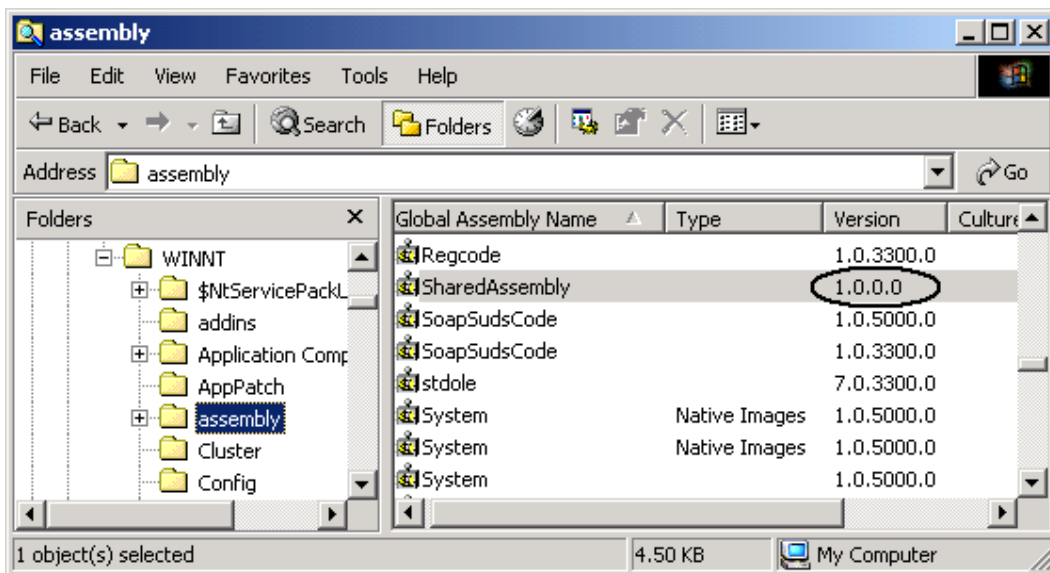
```
public VWMiniVan()  
{  
    MessageBox.Show("Using version 1.0.0.0!", "Shared Car");  
}
```

Kế tiếp, bạn cho nhật tu attribute **AssemblyVersion** trên tập tin AssemblyInfo.cs, cho sử dụng phiên bản 1.0.0.0 (như ta đã thấy trong phần đi trước). Bạn cho Build lại dự án

Việc bạn cần làm kế tiếp là bảo đảm lớp **SharedAssembly.dll** nguyên thủy sẽ bị gỡ bỏ khỏi GAC (bằng cách chọn mục **Delete** trên trình đơn shortcut khi bạn double-click lên tên assembly khi đang ở cửa sổ GAC. Tiếp theo, bạn cho di chuyển assembly 1.0.0.0 hiện hữu vào một thư mục con mới mang tên **Version1** chẳng hạn để bảo đảm là phiên bản này bị đông cứng. Xem hình 3-36.



Hình 3-36: Bảo toàn phiên bản 1.0.0.0



Hình 3-37: Cho SharedAssembly.dll trở về GAC lại

Bây giờ, một lần nữa ta cho assembly này trở về lại GAC, và bạn nhận ra rằng lần này phiên bản là <1.0.0.0>. Xem hình 3-37.

Một khi version 1.0.0.0 của SharedAssembly được đưa vào lại GAC, bạn right-click assembly này rồi click mục chọn **Properties** từ trình đơn shortcut. Bạn cho kiểm tra lỗi

tim về SharedAssembly này ánh xạ về thư mục con của Version1. Cuối cùng, bạn cho build lại và chạy ứng dụng **SharedAssemblyUser**. Và ứng dụng tiếp tục chạy ngon lành, nhưng lần này với một message cho biết phiên bản 1.0.0.0.

### 3.5.7.2 Xây dựng SharedAssembly, Version 2.0

Để minh họa cơ chế phiên bản trên .NET, ta thử thay đổi dự án **SharedAssembly** hiện hành. Bạn cho nhật tu lớp **VWMiniVan** với một thành viên mới (sử dụng enumeration) cho phép người sử dụng chọn những bản nhạc tiên tiến hơn. Ngoài ra, cũng phải nhật tu thông điệp được hiển thị trong lòng hàm constructor. Xem thí dụ 3-10:

#### Thí dụ 3-10: SharedAssembly, Version 2.0

```
*****
// Bạn muốn ban nhạc nào?
public enum BandName
{
    TonesOnTail,
    SkinnyPuppy,
    deftones,
    PTP
}

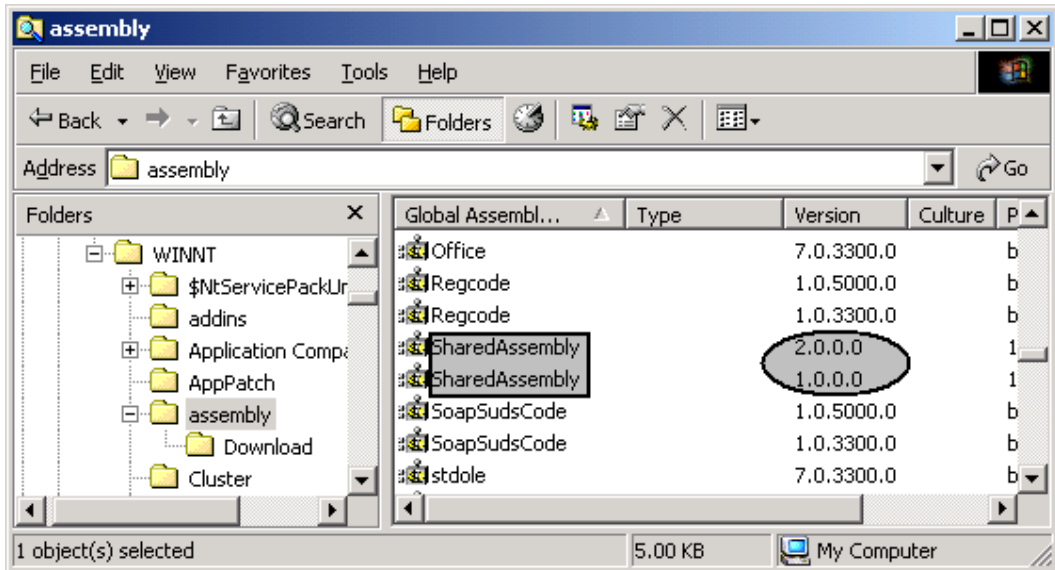
public class VWMiniVan
{
    public VWMiniVan()
    {
        MessageBox.Show("Using version 2.0.0.0!", "Shared Car");
    }
    . . .
    public void CrankGoodTones(BandName band)
    {
        switch(band)
        {
            case BandName.deftones:
                MessageBox.Show("So forget about me..."); break;
            case BandName.PTP:
                MessageBox.Show("Tick tick tock..."); break;
            case BandName.SkinnyPuppy:
                MessageBox.Show("Water vapor, to air..."); break;
            case BandName.TonesOnTail:
                MessageBox.Show("Ooooooh the rain. Oh the rain."); break;
            default:break;
        }
    }
}
*****
```

Trước khi cho biên dịch, bạn cho nâng cấp phiên bản này của assembly thành 2.0.0.0

```
// Cho nhật tu tập tin AssemblyInfo.cs
[assembly: AssemblyVersion("2.0.0.0")]
```

Nếu bạn nhìn vào thư mục debug của dự án, bạn sẽ thấy là bạn có một phiên bản mới của assembly SharedAssembly.dll, trong khi phiên bản đi trước nằm an toàn trong thư mục con Version1. Cuối cùng bạn cho cài đặt assembly mới này vào GAC. Như vậy giờ

đây bạn có hai phiên bản của cùng assembly **SharedAssembly.dll**, như theo hình 3-38.



**Hình 3-38: Thi hành tay trong tay (side-by-side)**

Bây giờ rõ ràng bạn đã có hai phiên bản của cùng một assembly được ghi nhận trên GAC, bạn có thể bắt đầu làm việc với các tập tin \*.config của ứng dụng để điều khiển việc kết nối với một phiên bản nào đó. Nhưng trước tiên, có đôi điều đối với cơ chế kết nối mặc nhiên đối với phiên bản.

### 3.5.7.3 *Tìm hiểu cơ chế kết nối mặc nhiên với phiên bản*

Như đã nói trước đây, nếu một ứng dụng client qui chiếu về một shared assembly, thì phiên bản major và minor phải giống nhau nếu muốn việc kết nối thành công. Tuy nhiên, .NET Runtime sẽ kết nối về một assembly nào đó nếu qui chiếu assembly khác biệt theo số build hoặc số revision. Hành động này được gọi là default version policy (cơ chế mặc nhiên kết nối về phiên bản), và được dùng để bảo đảm là bao giờ ứng dụng client cũng nhận được service release chót nhất và ngon lành nhất (nghĩa là được sửa chữa khỏi bug) của một assembly nào đó. Do đó, nếu manifest của ứng dụng client yêu cầu rõ ra phiên bản 1.0.0.0, nhưng GAC lại có phiên bản mới nhất bằng cách khai báo một QFE (chẳng hạn 1.0.2.2) thì ứng dụng client sẽ tự động nhận phiên bản được sửa chữa gần đây nhất. Theo thể thức này, ứng dụng client được bảo đảm là assembly được qui chiếu là tương thích lui và ngoài ra được bóc đi khỏi bug.

## 3.5.8 Khai báo cơ chế phiên bản custom

Khi bạn mong muốn điều khiển năng động cách một ứng dụng được gắn kết với một assembly thế nào, bạn cần viết ra một tập tin cấu hình ứng dụng. Các tập tin cấu hình là những khối lệnh XML dùng customize tiến trình gắn kết. Các tập tin này mang cùng tên ứng dụng chủ sở hữu kèm theo cái đuôi \*.config, đồng thời được đặt nằm ở thư mục ứng dụng. Ngoài tag `privatePath` (dùng cho biết dò tìm ở đâu đối với các private assembly), một tập tin cấu hình có thể khai báo thông tin liên quan đến shared assembly.

Điểm đầu tiên đáng quan tâm là sử dụng tập tin cấu hình ứng dụng để khai báo một phiên bản assembly cụ thể cần phải nạp vào, không cần biết đến ở manifest tương ứng được kê khai cái gì. Khi bạn muốn một ứng dụng client chuyển hướng kết nối về một shared assembly alternate, bạn sẽ dùng đến attribute **<dependentAssembly>** và **<bindingRedirect>**. Thí dụ, tập tin cấu hình sau đây sẽ ép sử dụng phiên bản 2.0.0.0:

```
<configuration>
  <runtime>
    <assemblyBinding xmlns="urn:schema-microsoft-com:asm.v1">
      <dependentAssembly>
        <assemblyIdentity name="sharedassembly"
          publicKeyToken="1b27a77f35a33c46"
          culture=""/>
        <bindingRedirect oldVersion="1.0.0.0"
          newVersion="2.0.0.0"/>
      </dependentAssembly>
    </assemblyBinding>
  </runtime>
</configuration>
```

Ở đây, tag `oldVersion` cho biết phiên bản bạn muốn cho phủ quyết, còn tag `newVersion` đúng là phiên bản bạn muốn cho nạp vào.

Để trải nghiệm, bạn tạo ra tập tin \*.config như trên và cho cất trữ vào thư mục của ứng dụng `SharedAssemblyUser`. Bạn bảo đảm là bạn đặt đúng tên tập tin cấu hình. Bây giờ bạn cho chạy chương trình bạn sẽ thấy là thông điệp cho biết “Using version 2.0.0.0”.

Nếu bạn nhật tu attribute `newVersion` thành 1.0.0.0, bạn sẽ thấy thông điệp sẽ là “Using version 1.0.0.0”.

Bạn thấy là có nhiều phiên bản cùng một assembly nằm “tay trong tay” trong GAC, và bạn có thể cấu hình cơ chế phiên bản như bạn muốn.

## Chương 4

# Tìm hiểu về Type Attributes và Reflection

Một ứng dụng .NET thường chứa đoạn mã, dữ liệu và metadata. *Metadata* là loại thông tin liên quan đến các kiểu dữ liệu, đoạn mã, assembly, v.v.. được trữ cùng với chương trình của bạn. Chương này sẽ khảo sát một vài metadata được tạo ra và được sử dụng.

*Attributes*<sup>11</sup> là một cơ chế dùng bổ sung metadata vào bản thân chương trình, chẳng hạn các chỉ thị trình biên dịch và các dữ liệu khác liên quan đến dữ liệu, hàm hành sự và lớp. Các attribute được đưa vào metadata và bạn có thể dùng trình tiện ích *ILDasm.exe* hoặc các công cụ đọc metadata để xem tận mắt metadata.

*Reflection* (phản chiếu) là tiến trình theo đây một chương trình có thể tự đọc những metadata riêng của mình. Một chương trình được gọi là “tự phản chiếu” khi nó trích metadata từ assembly và sử dụng metadata để hoặc thông tri cho người sử dụng biết hoặc để thay đổi các hành xử của mình.

## 4.1 Tìm hiểu về Attributes

Ngôn ngữ meta chính thức của COM (Component Object Model) là IDL (Interface Definition Language). IDL được dùng để mô tả tập hợp những kiểu dữ liệu được định nghĩa trong một COM server cổ điển nào đó. Để có thể mô tả một cách không nhập nhằng những kiểu dữ liệu, IDL đã dùng đến những “attributes”. Đây đơn giản chỉ là những từ chốt IDL đặt nằm trong cặp dấu [ ] (square bracket).

Vì IDL attribute tỏ ra rất hữu ích, nên C# (cũng như các ngôn ngữ ăn ý với .NET) cũng đưa vào trong ngôn ngữ C#. Do đó, một *attribute* được xem như là một đối tượng tượng trưng cho dữ liệu mà bạn muốn gắn liền với một phần tử nào đó trong chương trình của bạn. Phần tử mà bạn muốn gắn liền một attribute được gọi là đích (target) của attribute. Thí dụ, attribute:

```
[NoIDispatch]
```

---

<sup>11</sup> Chúng tôi không dịch là thuộc tính, mà để yên.

được gắn liền với một lớp hoặc một giao diện cho biết lớp target phải được dẫn xuất từ **IUnknown** thay vì **IDispatch**, khi xuất khẩu qua COM. COM sẽ được đề cập đến ở chương 7, “Tương tác với unmanaged code”.

Trong chương 3, “Tìm hiểu về Assembly và cơ chế Version” đi trước, bạn đã làm quen với attribute:

```
[assembly: AssemblyKeyFile(@"D:\Thien\theKey.snk")]
```

Attribute này chèn metadata vào assembly cho biết **StrongName** của chương trình.

## 4.1.1 Attribute “bẩm sinh” (intrinsic attribute)

Nếu bạn khảo sát các .NET namespace bạn sẽ thấy vô số attribute được định sẵn được hỗ trợ bởi CLR, và được gọi là là *intrinsic attribute* (*attribute bẩm sinh*). Bạn có thể sử dụng những attribute này trong ứng dụng của bạn. Ngoài ra, bạn cũng có thể tạo ra những attribute “cây nhà lá vườn” được gọi là *custom attribute* nếu bạn thấy cần thiết cải tiến cách hành xử của các lớp của bạn. Phần lớn lập trình viên chỉ sử dụng intrinsic attribute, mặc dù custom attribute cũng có thể là những công cụ cực mạnh nếu được phối hợp sử dụng với reflection, mà chúng tôi sẽ mô tả vào cuối chương này. Bạn nhớ cho là .NET attribute (intrinsic hoặc custom) hiện là những lớp được dẫn xuất từ **System.Attribute**, giúp nói rộng **System.Attribute** (trong khi IDL chỉ xem attribute như là những từ chốt đơn giản).

### 4.1.1.1 Attribute Target

Nếu bạn rảo xem qua CLR, bạn sẽ tìm thấy vô số attribute. Một vài attribute được áp dụng đối với assembly, một số khác đối với lớp hoặc interface, và một số khác, chẳng hạn **[WebMethod]** đối với thành viên lớp. Những attribute này được gọi là *attribute target*, được khai báo trong enumeration **AttributeTargets**. Bảng 4-01 cho thấy những attribute target.

**Bảng 4-01: Các attribute target có thể có được**

Tên thành viên	Sử dụng
<b>All</b>	Được áp dụng đối với bất cứ những phần tử sau đây: assembly, class, classmember, delegate, enum, event, field, interface, method, module, parameter, property, return value hoặc struct.
<b>Assembly</b>	Được áp dụng đối với bản thân assembly.
<b>Class</b>	Được áp dụng đối với những thể hiện của lớp.
<b>ClassMembers</b>	Được áp dụng đối với class, struct, enum, constructor, method,



	property, field, event, delegate và interface.
<b>Constructor</b>	Được áp dụng đối với một hàm constructor nào đó.
<b>Delegate</b>	Được áp dụng đối với một hàm hành sự được ủy thác
<b>Enum</b>	Được áp dụng đối với một liệt kê (enumeration)
<b>Event</b>	Được áp dụng đối với một tình huống
<b>Field</b>	Được áp dụng đối với một vùng mục tin
<b>Interface</b>	Được áp dụng đối với một giao diện
<b>Method</b>	Được áp dụng đối với một hàm hành sự
<b>Module</b>	Được áp dụng đối với đơn nguyên đơn lẻ
<b>Parameter</b>	Được áp dụng đối với thông số của một hàm hành sự
<b>Property</b>	Được áp dụng đối với một thuộc tính (cả hai get và set nếu được thi công)
<b>ReturnValue</b>	Được áp dụng đối với một trị trả về
<b>Struct</b>	Được áp dụng đối với cấu trúc (struct)

Các trị trên sẽ được trao qua như là thông số đối với attribute **AttributeUsage**. Attribute intrinsic này được dùng bởi trình biên dịch C# để tăng cường việc kiểm tra sử dụng đúng đắn một custom attribute. Bạn sẽ thấy cách sử dụng attribute intrinsic **AttributeUsage** và enumeration **AttributeTargets** trên mục 4.1.2.1, “Khai báo một attribute”.

#### 4.1.1.2 Áp dụng các attribute

Bạn áp dụng các attribute đối với những target của chúng bằng cách đặt chúng vào những cặp dấu ngoặc vuông [ ] trước mục target. Nếu bạn muốn áp dụng nhiều attribute lên một attribute target thì bạn có thể phối hợp các attribute này, bằng cách hoặc viết chồng cái này lên cái kia:

```
[assembly: AssemblyDelaySignAttribute(false)]
[assembly: AssemblyKeyFileAttribute(".\\keyFile.snk")]
```

hoặc phân cách các attribute bởi dấu phẩy:

```
[assembly: AssemblyDelaySignAttribute(false),
assembly: AssemblyKeyFileAttribute(".\\keyFile.snk")]
```

**Bạn để ý:** Bạn phải đặt assembly attribute sau các lệnh using và trước bất cứ đoạn mã nào.

Nhiều intrinsic attribute được sử dụng để hoạt động liên hoàn (interoperating) với COM, như sẽ được đề cập đến ở chương 7, “Tương tác với unmanaged code”). Bạn đã làm quen với các attribute **[Serializable]** và **[NonSerialized]** khi đề cập đến xuất nhập dữ liệu ở chương 1, “Xuất nhập dữ liệu & Sản sinh hằng loạt đối tượng”.

Namespace **System.Runtime** cung cấp một số intrinsic attribute, bao gồm những attribute đối với assembly (chẳng hạn attribute **keyname**), hoặc đối với cấu hình (chẳng hạn **debug** cho biết debug build), cũng như đối với phiên bản.

Bạn có thể tổ chức các intrinsic attribute theo cách các attribute này được sử dụng thế nào. Các intrinsic attribute chủ yếu là những attribute được sử dụng bởi COM, hoặc những attribute dùng để điều chỉnh tập tin IDL từ trong lòng một tập tin mã nguồn, hoặc những attribute được sử dụng bởi các lớp ATL Server, và những attribute dùng bởi trình biên dịch Visual C++.

Có thể attribute mà bạn sử dụng nhiều nhất trong việc lập trình C# hằng ngày (nếu bạn không tương tác với COM) là **[Serializable]**. Như bạn có thể thấy trong chương 1, "Xuất nhập dữ liệu & Sản sinh hàng loạt đối tượng", để bảo đảm là lớp của bạn có thể được serialized lên đĩa hoặc lên Web, bạn chỉ cần thêm attribute **[Serializable]** lên lớp:

```
[Serializable]
class MySerializableClass
```

Tag attribute sẽ được đặt nằm trong cặp dấu ngoặc vuông [ ] ngay liền trước attribute target - trong trường hợp trên là định nghĩa lớp.

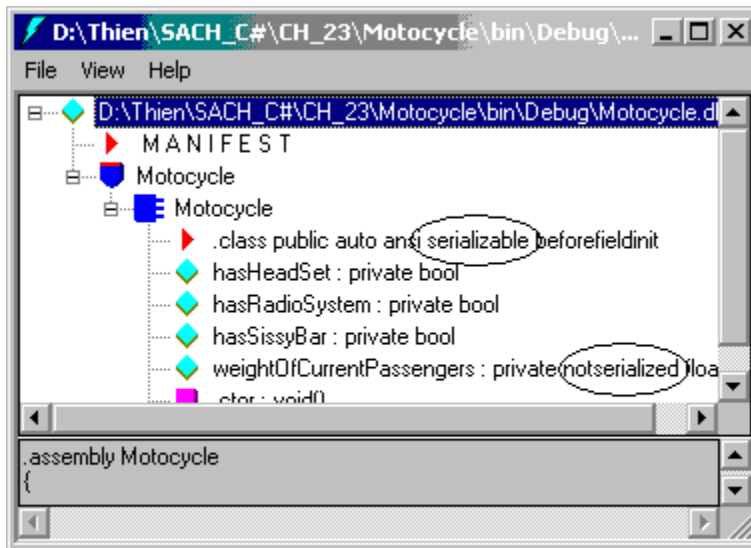
Thí dụ, giả sử bạn muốn gán attribute **[Serializable]** cho một item nào đó, lớp Motorcycle sau đây, như theo thí dụ 4-01, gán một attribute cho bản thân lớp. Nhưng nếu có một item nào đó bạn không muốn trữ lên đĩa thì bạn dùng attribute **[NonSerialized]**

### ***Thí dụ 4-01: Sử dụng attribute đối với một lớp***

```
// Lớp này có thể được cho trữ lên đĩa
[Serializable]
public class Motorcycle
{
    bool hasRadioSystem;
    bool hasHeadSet;
    bool hasSissyBar;

    // Nhưng khi trữ, thì khỏi trữ vùng mục tin này
    [NonSerialized]
    float weightOfCurrentPassengers;
}
```

Nếu bạn sử dụng trình tiện ích **ILDasm.exe** để khảo sát, bạn sẽ thấy là các attribute này giờ đây được khai báo trong lòng type metadata. Xem hình 4-01.



Hình 4-01: Attribute được tương trưng bởi metadata

## 4.1.2 Custom Attribute

C# cho phép bạn tạo những custom attribute và sử dụng chúng vào lúc chạy nếu bạn muốn. Bạn nhớ lại, kỳ thật attribute là một lớp được dẫn xuất từ **System.Attribute**. Do đó, khi bạn áp dụng attribute **[Serializable]** cho lớp **Motocycle**, thì thật ra ta áp dụng một thể hiện của kiểu dữ liệu **System.Serializable**. Nhìn theo khía cạnh thiết kế, một attribute là một thể hiện lớp (class instance) có thể được đem áp dụng đối với một kiểu dữ liệu khác. Trong thế giới lập trình thiên đối tượng, kiểu tiếp cận này được gọi là *aspect-oriented programming* (lập trình thiên khía cạnh).

Giả sử, thí dụ tổ chức của bạn muốn theo dõi việc sửa chữa những bug. Bạn đã có sẵn một căn cứ dữ liệu ghi nhận tất cả các bug, nhưng bạn muốn gắn kết những bản báo cáo về những sửa chữa cụ thể trong đoạn mã. Có thể bạn thêm những chú giải vào đoạn mã như sau:

```
// Bug 323 được sửa chữa bởi Nguyễn Xuân Dũng ngày 1/1/2004
```

Điều này thì dễ thấy trong đoạn mã nguồn, nhưng lại không có một kết nối với Bug 323 trên căn cứ dữ liệu. Đây chính là lúc một custom attribute vào cuộc giúp bạn. Bạn có thể thay thế comment bởi một cái gì như sau:

```
[BugFixAttribute(323, "Nguyễn Xuân Dũng", "1/1/2004")  
Comment="Off by one error"]
```

Sau đó, bạn có thể viết một chương trình cho đọc metadata để tìm ra tất cả các ghi chú liên quan đến sửa chữa bug và cho nhật tu căn cứ dữ liệu. Attribute có thể dùng vào mục đích chú giải nhưng cũng cho phép bạn tìm lại những thông tin theo lập trình thông qua những công cụ mà bạn tạo ra.

### 4.1.2.1 Khai báo một attribute

Bước đầu tiên trong việc tạo một custom attribute là khai báo một lớp mới được dẫn xuất từ **System.Attribute**. Theo qui ước đặt tên, bạn phải thêm cái đuôi “Attribute” vào kiểu dữ liệu mới. Thí dụ, **BugFixAttribute** chẳng hạn :

```
public class BugFixAttribute: System.Attribute
```

Như vậy, ở đây **BugFixAttribute** cho phép lập trình viên đưa vào một chuỗi vào type metadata mô tả một sửa chữa bug đặc biệt.

Bạn cần cho trình biên dịch biết loại phân tử nào attribute này có thể được sử dụng (attribute target). Bạn khai báo điều này với một attribute:

```
[AttributeUsage(AttributeTargets.Class |  
    AttributeTargets.Constructor |  
    AttributeTargets.Field |  
    AttributeTargets.Method |  
    AttributeTargets.Property,  
    AllowMultiple = true)]
```

**AttributeUsage** là một attribute được áp dụng đối với các attribute: một loại meta-attribute. Nó cung cấp nếu bạn muốn meta-metadata, nghĩa là dữ liệu liên quan đến metadata. Đối với attribute constructor **AttributeUsage**, bạn trao qua hai đối mục. Đối mục thứ nhất là một lô những flag cho biết target - trong trường hợp này là class, constructor, field, method và property, còn đối mục thứ hai cho biết phân tử chỉ định có thể nhận nhiều hơn một attribute như thế. Trong thí dụ này, **AllowMultiple** được cho về **true** cho biết các thành viên lớp có thể có nhiều hơn một **BugFixAttribute** được gán cho.

### 4.1.2.2 Đặt tên cho một attribute

Attribute “cây nhà lá vườn” mới trong thí dụ này là **BugFixAttribute**. Qui ước đặt tên là cho gán cái đuôi “Attribute” vào tên attribute của bạn. Trình biên dịch chấp nhận việc bạn được phép triệu gọi attribute với tên ngắn gọn (nghĩa là không có cái đuôi “Attribute”). Do đó, bạn có thể viết:

```
[BugFix(323, “Nguyễn Xuân Dũng”, “1/1/2004”)  
    Comment=“Sửa chữa sai lầm”]
```

Trước tiên, trình biên dịch sẽ đi tìm một attribute mang tên **BugFix**, và nếu tìm không thấy sẽ tìm tiếp **BugFixAttribute**.

### 4.1.2.3 Xây dựng một attribute

Mỗi attribute phải có ít nhất một hàm constructor. Attribute sẽ nhận hai loại thông số: loại dựa theo vị trí (*positional*) và loại dựa theo tên *named*. Trên thí dụ **BugFix**, tên lập trình viên và ngày tháng năm là những thông số vị trí, còn **comment** là thông số theo tên. Các thông số theo vị trí sẽ được chuyển giao thông qua hàm constructor và phải được trao theo thứ tự được khai báo trong hàm constructor:

```
// hàm constructor
public BugFixAttribute(int bugID, string programmerName, string date)
{
    this.bugID = bugID;
    this.programmerName = programmerName;
    this.date = date;
}
```

Các thông số theo tên sẽ được thi công như là những thuộc tính:

```
public string Comment
{
    get { return comment; }
    set { comment = value; }
}
```

Đối với các thông số positional, phổ biến là người ta tạo những thuộc tính read-only:

```
public int BugID
{
    get { return bugID; }
}
```

### 4.1.2.4 Sử dụng một attribute

Một khi bạn đã định nghĩa một attribute, bạn có thể đưa nó vào sử dụng bằng cách đặt nó nằm ngay liền trước target của nó. Muốn trải nghiệm **BugFixAttribute** của thí dụ đi trước, chương trình sau đây tạo một lớp đơn giản mang tên **MyMath** và đưa ra hai hàm. Bạn sẽ gán **BugFixAttributes** đối với lớp để ghi nhận việc theo dõi các bug theo thời gian.

```
[BugFixAttribute(121, "Nguyễn Xuân Dũng", "01/03/2004")]
[BugFixAttribute(107, "Nguyễn Xuân Dũng", "01/04/2004",
    Comment="Sửa chữa sai lầm")]
public class MyMath
```

Các attribute này sẽ được trữ với metadata. Thí dụ, 4-02 sau đây cho thấy toàn bộ chương trình:

***Thí dụ, 4-02: Làm việc với custom attribute***

```
using System;
using System.Reflection;

namespace TestAttribute
{
    public class Tester
    {
        public static void Main()
        {
            MyMath mm = new MyMath();
            Console.WriteLine("Calling DoFunc(7).Result: {0}",
                             mm.DoFunc1(7));
            Console.ReadLine();
        }
    }

    //Tạo custom attribute dùng gán cho các thành viên lớp
    [AttributeUsage(AttributeTargets.Class |
        AttributeTargets.Constructor |
        AttributeTargets.Field |
        AttributeTargets.Method |
        AttributeTargets.Property,
        AllowMultiple = true)]
    public class BugFixAttribute: System.Attribute
    {
        // Attribute constructor đối với positional parameters
        public BugFixAttribute(int bugID, string programmerName,
                               string date)
        {
            this.bugID = bugID;
            this.programmerName = programmerName;
            this.date = date;
        }

        // Các hàm accessor
        public string Comment
        {
            get { return comment; }
            set { comment = value; }
        }

        public int BugID
        {
            get { return bugID; }
        }

        public string Date
        {
            get { return date; }
        }

        public string ProgrammerName
        {
            get { return programmerName; }
        }
    }
}
```

```

    }

    // Các dữ liệu thành viên
    private int bugID;
    private string comment;
    private string date;
    private string programmerName;
}

// **** gán các attribute cho lớp ****
[BugFixAttribute(121, "Nguyễn Xuân Dũng", "01/03/2004")]
[BugFixAttribute(107, "Nguyễn Xuân Dũng", "01/04/2004",
    Comment="Fixed off by one errors")]
public class MyMath
{
    public double DoFunc1(double param1)
    {
        return param1 + DoFunc2(param1);
    }

    public double DoFunc2(double param1)
    {
        return param1 / 3;
    }
}
}

```

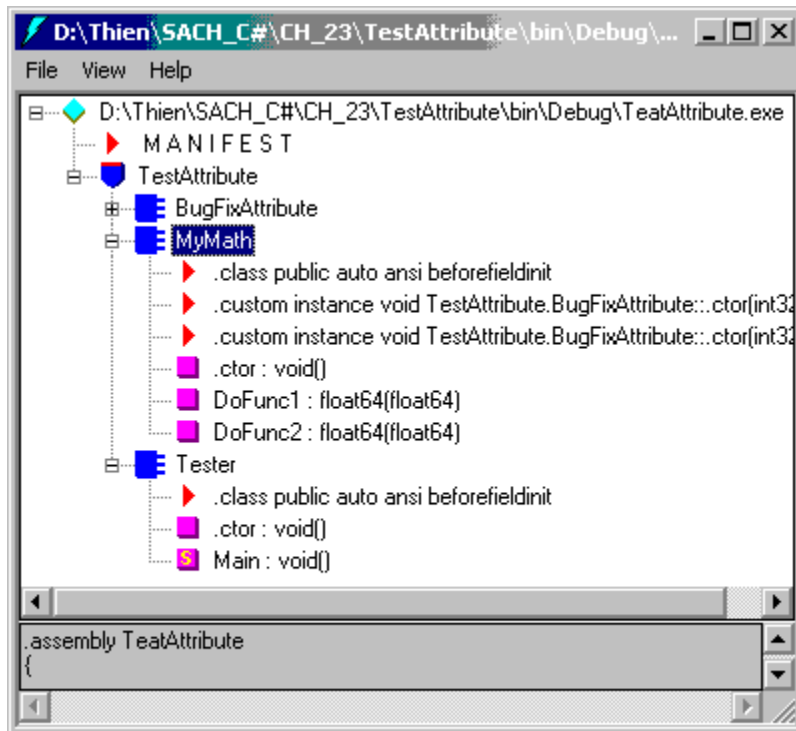
Hình 4-02 cho thấy kết xuất



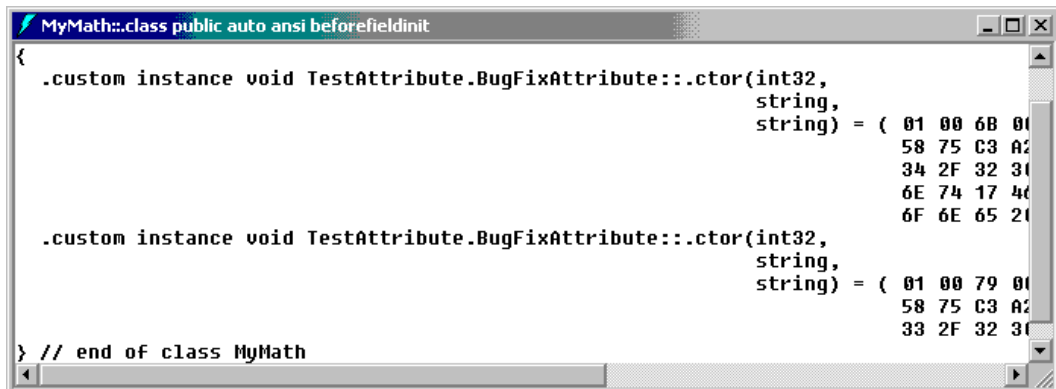
Hình 4-02: Kết xuất TestAttribute

Như bạn có thể thấy, các attribute tuyệt đối không ảnh hưởng lên kết xuất. Hiện thời bạn cứ tin là các attribute hiện hữu. Nếu bạn sử dụng ILDasm.exe bạn có thể thấy metadata của assembly TestAttribute như theo hình 4-03 dưới đây.

Ngoài ra, cũng trên ILDasm.exe bạn có thể cho hiển thị thể hiện của **BugFixAttribute** hình 4-04 là phần A, và hình 4-05 là phần B. Trên phần B bạn thấy chuỗi dữ liệu hình thành attribute. Chúng ta sẽ xem cách đi lấy metadata này và sử dụng metadata trong chương trình của bạn trong phần kế tiếp.

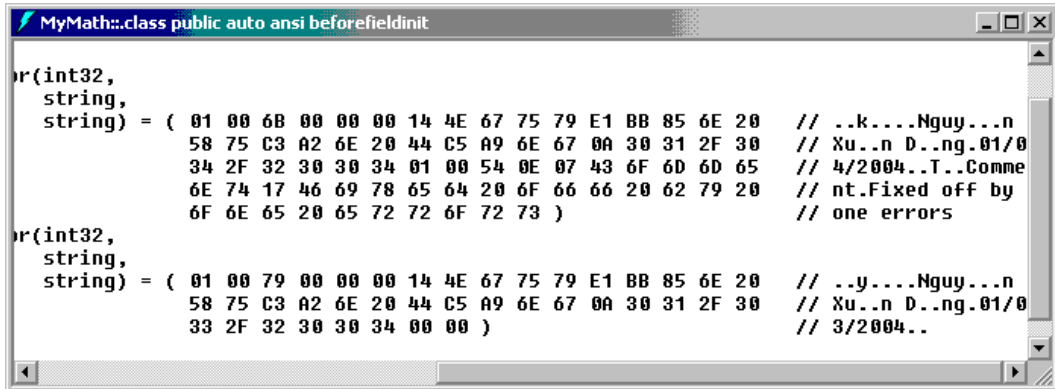


Hình 4-03: Metadata trong assembly



Hình 4-04: Custom Instance của BugFixAttribute - phần A





Hình 4-05: Custom instance của BugFixAttribute - phần B

### 4.1.3 Assembly Level Attributes và Module Level Attributes

Ta có khả năng áp dụng attribute đối với tất cả các kiểu dữ liệu trong lòng một module hoặc đối với tất cả các module thuộc một assembly nào đó. Muốn thế, ta phải sử dụng đến bất cứ các attribute intrinsic nào, mà attribute được sử dụng nhiều nhất là [assembly:] và [module:].

Thí dụ, giả sử bạn mong muốn bảo đảm là tất cả các kiểu dữ liệu được định nghĩa trong assembly của bạn là “chiều theo ý” (compliant) của CLS (Common Language Specification).

```
// Tăng cường CLS Compliance!
using System;
[assembly: System.CLSCompliantAttribute(true)]

namespace MyAttributes
{
    [VehiculeDescriptionAttribute("A very long,
                                   slow but feature rich auto")]
    public class Winnebago
    {
        public Winnebago() {}
    }
}
```

Bây giờ nếu ta thêm một dòng lệnh nằm ngoài đặc tả CLS như sau:

```
// Kiểu dữ liệu Ulong không ăn khớp với CLS
public class Winnebago
{
```

```
public Winnebago() {}

public ulong notCompliant;
}
```

trình biên dịch sẽ tung ra sai lầm “*Type of “MyAttributes.Winnebago.notCompliant” is not a CLS-compliant*”.

.NET attribute [CLSCompliant] tương đương với attribute IDL [oleautomation], bảo đảm là tất cả các thành viên giao diện là VARIANT compliant, và như vậy có thể nhận biết bởi một ngôn ngữ ăn ý với COM.

Bạn để ý cú pháp [assembly:] được dùng để báo cho trình biên dịch biết attribute CLSCompliant phải được áp dụng đối với cấp độ assembly chứ không được áp dụng đối với một kiểu dữ liệu đơn lẻ trong lòng assembly chẳng hạn. Một điểm đáng ghi nhận là [assembly:] và [module:] phải được đặt *nằm ngoài* của một định nghĩa namespace.

## 4.1.4 Tập tin Studio.NET AssemblyInfo.cs

Các dự án Visual Studio .NET đều định nghĩa một tập tin mang tên **AssemblyInfo.cs**. Tập tin này rất tiện lợi để đặt những attribute cần được áp dụng ở cấp assembly. Bảng 4-02 liệt kê một vài assembly level attribute mà bạn nên quan tâm:

**Bảng 4-02: Các assembly level attribute được chọn lọc**

Assembly-Level Attribute	Mô tả
AssemblyCompanyAttribute	Nơi trữ những thông tin cơ bản của công ty
AssemblyConfigurationAttribute	Thông tin liên quan đến build, chẳng hạn “retail” hoặc “debug”
AssemblyCopyrightAttribute	Nơi trữ bất cứ thông tin bản quyền đối với sản phẩm hoặc assembly
AssemblyDescriptionAttribute	Mô tả ngắn gọn sản phẩm hoặc module hình thành assembly
AssemblyInformationalVersionAttribute	Thông tin bổ sung hoặc hỗ trợ về phiên bản, chẳng hạn số phiên bản sản phẩm kinh doanh
AssemblyProductAttribute	Thông tin liên quan đến sản phẩm
AssemblyTrademarkAttribute	Thông tin liên quan đến thương hiệu
AssemblyCultureAttribute	Thông tin liên quan đến nền văn hóa hoặc ngôn ngữ mà assembly chịu hỗ trợ
AssemblyKeyFileAttribute	Cho biết tên tập tin chứa cặp mục khóa (key pair) dùng đánh dấu assembly (nghĩa là dùng tạo một strong name)

<b>AssemblyOperationngSystemAttribute</b>	Thông tin liên quan đến hệ điều hành nào assembly đang được build sẽ hỗ trợ.
<b>AssemblyProcessorAttribute</b>	Thông tin liên quan đến bộ xử lý nào assembly đang được build sẽ hỗ trợ.
<b>AssemblyVersionAttribute</b>	Cho biết thông tin phiên bản của assembly theo dạng thức major:minor:build:revision”

## 4.1.5 Khám phá các attributes vào lúc chạy

Như bạn có thể thấy, ta có khả năng nhận những attribute vào lúc chạy bằng cách dùng lớp **Type**, như theo thí dụ 4-03 sau đây:

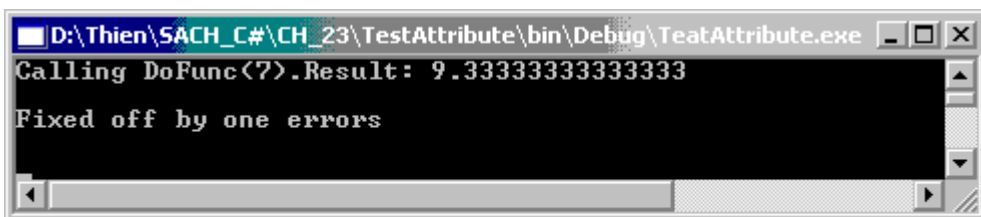
### Thí dụ 4-03: Đọc các attribute

```
public class AttReader
{
    public static int Main(string[] args)
    {
        // Đi lấy kiểu dữ liệu của MyMath
        Type t = typeof(MyMath);

        // Đi lấy tất cả các attribute trong assembly
        object[] customAtts = t.GetCustomAttributes(false);

        // Liệt kê tất cả các thông tin
        foreach (BugFixAttribute v in customAtts)
            Console.WriteLine(v.Comment);
        return 0;
    }
}
```

Hình 4-06 cho thấy kết xuất:



Hình 4-06: Đọc custom attribute của Bạn

## 4.2 Tìm hiểu về Reflection

Muốn cho các attribute trong metadata thành hữu ích, bạn cần có cách truy xuất được chúng, nhất là trong lúc chạy. Các lớp trong namespace **Reflection**, cùng với các lớp thuộc **System.Type** và **System.TypeReference**, sẽ giúp bạn quan sát và tương tác với metadata. Trong thế giới .NET, *reflection* là tiến trình khám phá kiểu dữ liệu vào lúc chạy. Sử dụng các dịch vụ của reflection, bạn có khả năng nạp một assembly vào lúc chạy và khám phá ra cùng loại thông tin giống như với **ILDasm.exe**. Thí dụ, bạn có thể có được một bảng liệt kê tất cả các kiểu dữ liệu nằm trong một module nào đó kể cả các hàm hành sự, vùng mục tin, thuộc tính và tình huống được định nghĩa đối với một kiểu dữ liệu nào đó. Một cách năng động, bạn cũng có thể phát hiện những giao diện mà một lớp (hoặc struct) nào đó chịu hỗ trợ, các thông số thuộc một hàm hành sự cũng như các chi tiết có liên hệ (base class, namespace information, v.v.).

Reflection thường được sử dụng đối với bất cứ một trong 4 công tác sau đây:

- **Nhìn xem metadata:** Việc này có thể được sử dụng bởi những công cụ cũng như trình tiện ích muốn hiển thị metadata.
- **Phát hiện kiểu dữ liệu:** Việc làm này cho phép bạn quan sát các kiểu dữ liệu trong một assembly, tương tác với chúng hoặc thể hiện các kiểu dữ liệu này. Điều này có thể hữu ích khi tạo ra những custom script. Thí dụ, có thể bạn muốn cho phép người sử dụng tương tác với chương trình của bạn bằng cách sử dụng một ngôn ngữ kịch bản (script language), chẳng hạn JavaScript hoặc một ngôn ngữ kịch bản bạn tự tạo ra.
- **Gắn kết trễ (late binding) với các hàm hành sự và thuộc tính** Điều này cho phép lập trình viên triệu gọi một cách năng động các hàm và thuộc tính đối với các đối tượng được thể hiện dựa trên việc phát hiện kiểu dữ liệu.
- **Tạo các kiểu dữ liệu vào lúc chạy (Reflection Emit).** Sử dụng cuối cùng của reflection là tạo ra những kiểu dữ liệu mới vào lúc chạy rồi cho sử dụng những kiểu dữ liệu này để thực hiện các công tác. Bạn có thể làm việc này khi một lớp custom, được tạo vào lúc chạy, sẽ chạy nhanh hơn so với đoạn mã generic được tạo vào lúc biên dịch. Chúng tôi sẽ đưa ra một thí dụ vào cuối chương.

Muốn hiểu các dịch vụ của **Reflection**, bạn cần hiểu sâu lớp **Type** (được định nghĩa trong namespace **System**) cũng như namespace **System.Reflection**. Như bạn sẽ thấy, lớp **System.Type** chứa một số hàm hành sự cho phép bạn truy ra những thông tin đáng giá liên quan đến kiểu dữ liệu hiện hành bạn đang quan sát. Còn **System.Reflection** thì lại chứa vô số kiểu dữ liệu liên hệ cho phép gắn kết trễ (late binding) cũng như nạp động các assembly. Bây giờ ta bắt đầu khảo sát lớp **System.Type**.

## 4.2.1 Type Class

Lớp **Type** là gốc rễ của các lớp reflection. Type gói ghém một biểu diễn kiểu dữ liệu của một đối tượng. Lớp Type là thể thức chủ yếu để truy cập metadata. Nhiều item được định nghĩa trong namespace **System.Reflection** sử dụng đến lớp **System.Type**. Lớp này cung cấp một số hàm hành sự mà bạn có thể dùng để phát hiện những chi tiết nằm sâu trong một item nào đó. Bảng 4-03 cho liệt kê một số thành viên chủ chốt của lớp **Type** này.

**Bảng 4-03: Các thành viên của lớp Type**

Các thành viên	Mô tả
<b>IsAbstract</b> <b>IsArray</b> <b>IsClass</b> <b>IsCOMObject</b> <b>IsEnum</b> <b>IsInterface</b> <b>IsPrimitive</b> <b>IsNestedPublic</b> <b>IsNestesPrivate</b> <b>IsSealed</b> <b>IsValueType</b>	Các thuộc tính này (trong số các thuộc tính khác) cho phép bạn khám phá một số nét cơ bản liên quan đến Type bạn đang qui chiếu (nghĩa là có phải là một hàm hành sự trừ tượng hoặc không, có phải là một bản dãy hay không, v.v..)
<b>GetConstructor()</b> <b>GetEvents()</b> <b>GetFields()</b> <b>GetInterfaces()</b> <b>GetMethods()</b> <b>GetMembers()</b> <b>GetNestedTypes()</b> <b>GetProperties()</b>	Các hàm hành sự này (trong số các hàm hành sự khác) cho phép bạn có được một bản dãy tượng trưng cho các item (interface, method, property, v.v..) mà bạn quan tâm. Mỗi hàm hành sự sẽ trả về một bản dãy liên đới (nghĩa là <b>GetFields()</b> sẽ trả về một bản dãy <b>FieldInfo</b> , <b>GetMethods()</b> sẽ trả về một bản dãy <b>MethodInfo</b> , v.v..). Bạn nên nhớ mỗi một những hàm hành sự này có một dạng thức đặc biệt (nghĩa là <b>GetMethod()</b> , <b>GetProperty()</b> cho phép bạn tìm lại một item cụ thể dựa theo tên thay vì một bản dãy của tất cả các item có dính dáng).
<b>FindMembers()</b>	Hàm này trả về một bản dãy cáac kiểu dữ liệu MemberInfo, dựa trên một tiêu chí truy tìm.
<b>GetType()</b>	Hàm này trả về một thể hiện Type cho ra một tên chuỗi.
<b>InvokeMember()</b>	Hàm này cho phép late binding đối với một item nào đó.

### 4.2.1.1 Nhận về một đối tượng Type

Có nhiều cách cho phép bạn nhận về một thể hiện của lớp **Type**. Tuy nhiên, có một điều bạn không thể làm được là trực tiếp tạo một đối tượng *Type* bằng cách sử dụng từ

chốt **new**, xem **Type** như là một lớp trừu tượng. Trước tiên, **System.Object** định nghĩa một hàm hành sự **GetType()** trả về một thể hiện của lớp **Type**:

```
// Truy ra Type sử dụng một thể hiện hợp lệ Foo
Foo theFoo = new Foo();
Type t = theFoo.GetType();
```

Ngoài cách kể trên, bạn cũng có thể nhận về một **Type** bằng cách dùng ngay bản thân lớp **Type**. Muốn thế, bạn cho triệu gọi hàm static **GetType()** và khai báo tên chuỗi của item mà bạn quan tâm:

```
// Truy ra Type sử dụng hàm hành sự static Type.GetType()
Type t = null;
t = Type.GetType("Foo");
```

Cuối cùng, bạn cũng có thể nhận về một thể hiện kiểu **Type** bằng cách dùng từ chốt **typeof()**:

```
// Truy ra Type bằng cách sử dụng từ chốt typeof()
Type t = typeof(Foo);
```

Bạn để ý **Type.GetType()** và **typeof()** chỉ hữu ích khi bạn không cần phải tạo trước tiên một thể hiện đối tượng để lấy thông tin liên quan đến kiểu dữ liệu. Bây giờ bạn đã có một qui chiếu về **Type**, ta thử xem qua cách sử dụng nó thế nào.

### 4.2.1.2 Sử dụng lớp *Type*

Để minh họa sự hữu ích của **System.Type**, giả sử bạn có một lớp mang tên **Foo** được định nghĩa như theo thí dụ 4-04 sau đây. Toàn bộ dự án được liệt kê sau đây.

#### **Thí dụ 4-04: Sử dụng *System.Type* thế nào**

```
using System;
using System.Reflection;

namespace TestType
{
    // Đây là những item mà ta sẽ phát hiện vào runtime

    // Hai giao diện
    public interface IFaceOne
    { void MethodA(); }

    public interface IFaceTwo
    { void MethodB(); }
```

```

// Lớp Foo hỗ trợ hai giao diện này
public class Foo: IFaceOne, IFaceTwo
{
    // Các vùng mục tin
    public int myIntField;
    public string myStringField;

    // Một hàm hành sự
    public void myMethod(int p1, string p2) { }

    // Một thuộc tính
    public int MyProp
    {
        get {return myIntField;}
        set {myIntField = value;}
    }

    // Các hàm IFaceOne và IFaceTwo
    public void MethodA() { }
    public void MethodB() { }
}

class FooReader
{
    // In ra vài thông kê lý thú liên quan đến Foo
    public static void ListVariousStats(Foo f)
    {
        Console.WriteLine("**** Various stats about Foo ****");

        Type t = f.GetType();
        Console.WriteLine("Full name is: {0}", t.FullName);
        Console.WriteLine("Base is: {0}", t.BaseType);
        Console.WriteLine("Is it abstract? {0}", t.IsAbstract);
        Console.WriteLine("Is it a COM object? {0}",
            t.IsCOMObject);
        Console.WriteLine("Is it sealed? {0}", t.IsSealed);
        Console.WriteLine("Is it a class? {0}", t.IsClass);

        Console.WriteLine("*****\n");
    }

    // In ra tất cả tên các hàm
    public static void ListMethods(Foo f)
    {
        Console.WriteLine("**** Methods of Foo ****");

        Type t = f.GetType();
        MethodInfo[] mi = t.GetMethods();
        foreach(MethodInfo m in mi)
            Console.WriteLine("Method: {0}", m.Name);
        Console.WriteLine("*****\n");
    }

    // In ra tất cả tên các vùng mục tin

```

```

public static void ListFields(Foo f)
{
    Console.WriteLine("**** Fields of Foo ****");

    Type t = f.GetType();
    FieldInfo[] fi = t.GetFields();
    foreach(FieldInfo field in fi)
        Console.WriteLine("Field: {0}", field.Name);
    Console.WriteLine("*****\n");
}

// Đi lấy tất cả các thuộc tính
public static void ListProp(Foo f)
{
    Console.WriteLine("**** Properties of Foo ****");

    Type t = f.GetType();
    PropertyInfo[] pi = t.GetProperties();
    foreach(PropertyInfo prop in pi)
        Console.WriteLine("Prop: {0}", prop.Name);
    Console.WriteLine("*****\n");
}

// Liệt kê ra tất cả các giao diện
public static void ListInterfaces(Foo f)
{
    Console.WriteLine("**** Interfaces of Foo ****");

    Type t = f.GetType();
    Type[] ifaces = t.GetInterfaces();
    foreach(Type i in ifaces)
        Console.WriteLine("Interface: {0}", i.Name);
    Console.WriteLine("*****\n");
}

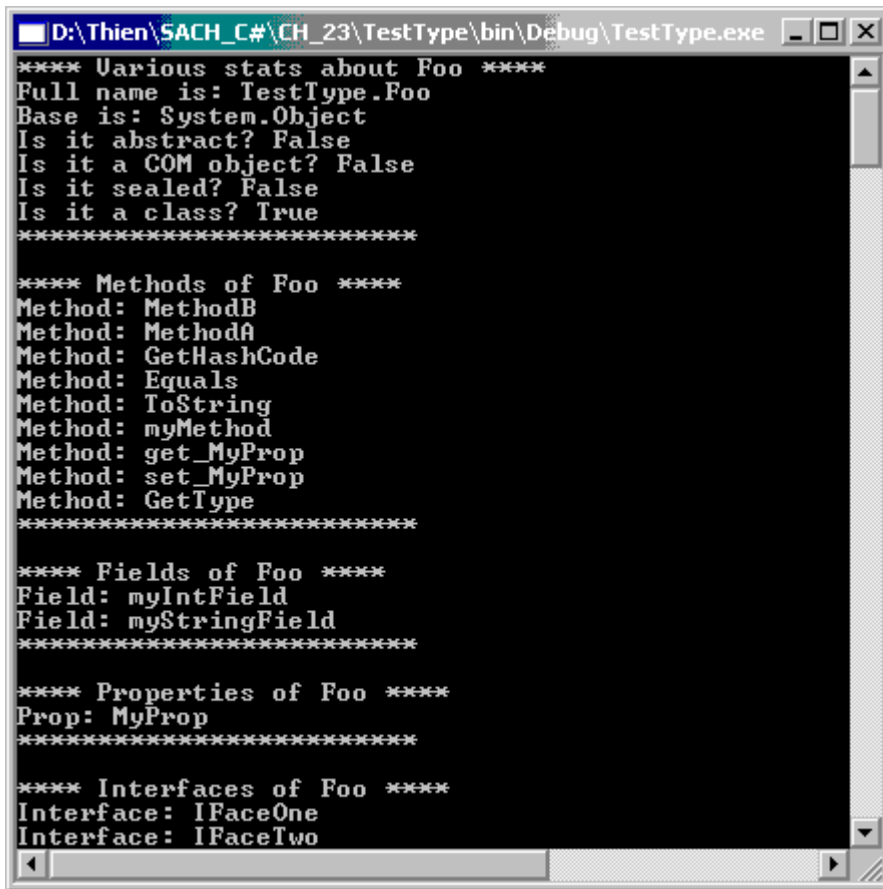
[STAThread]
public static int Main(string[] args)
{
    Foo theFoo = new Foo();
    ListVariousStats(theFoo);
    ListMethods(theFoo);
    ListFields(theFoo);
    ListProp(theFoo);
    ListInterfaces(theFoo);

    Console.ReadLine();
    return 0;
}
}

```

Hình 4-07 cho thấy kết xuất của dự án TestType.





```
D:\Thien\SACH_C#\CH_23\TestType\bin\Debug\TestType.exe
**** Various stats about Foo ****
Full name is: TestType.Foo
Base is: System.Object
Is it abstract? False
Is it a COM object? False
Is it sealed? False
Is it a class? True
*****

**** Methods of Foo ****
Method: MethodB
Method: MethodA
Method: GetHashCode
Method: Equals
Method: ToString
Method: myMethod
Method: get_MyProp
Method: set_MyProp
Method: GetType
*****

**** Fields of Foo ****
Field: myIntField
Field: myStringField
*****

**** Properties of Foo ****
Prop: MyProp
*****

**** Interfaces of Foo ****
Interface: IFaceOne
Interface: IFaceTwo
```

Hình 4-07: Kết xuất của TestType dựa vào Reflection.

Như bạn thấy ở trên lớp Foo hỗ trợ hai giao diện **IFaceOne** và **IFaceTwo**. Tạm thời trong thí dụ này việc thi công hai hàm MethodA và MethodB không quan trọng nên không ghi ra đây làm gì.

Tiếp theo là một lớp, **FooReader**, có khả năng khám phá các method, property, hỗ trợ interface và fields đối với một đối tượng Foo. Lớp **FooReader** định nghĩa một số hàm static gần giống như nhau. Các hàm static mang tên ListVariousStats(), ListMethods(), ListFields(), ListProps(), và ListInterfaces() liệt kê các item theo loại thể nào. Các hàm này khá rõ ràng khỏi cần giải thích chi thêm. Cuối cùng hàm Main() chỉ làm phận sự tuần tự triệu gọi các hàm static này vào. Kết quả là hình 4-07.

## 4.2.2 Nhìn xem Metadata

Trong phần này, bạn sẽ sử dụng sự hỗ trợ của C# Reflection để đọc metadata trong lớp **MyMath** (xem thí dụ 4-02, mục 4.1.2.4). Tuy nhiên, trước khi đi sâu chúng ta nên khảo sát qua namespace **System.Reflection**.

### 4.2.2.1 Khảo sát namespace *System.Reflection*

Giống như bất cứ namespace nào, **System.Reflection** chứa một số kiểu dữ liệu có liên hệ với nhau, kiểu này bạn sẽ quan tâm hơn kiểu khác. Bảng 4-04 sẽ liệt kê một số kiểu dữ liệu cốt lõi mà bạn nên quan tâm, một số bạn đã làm quen trong thí dụ Foo đi trước.

**Bảng 4-04: Các thành viên cốt lõi của namespace *System.Reflection***

Các thành viên	Mô tả
<b>Assembly</b>	Lớp này (cộng thêm vô số kiểu có liên hệ) chứa một số hàm hành sự cho phép bạn nạp một assembly, khảo sát và thao tác assembly.
<b>AssemblyName</b>	Lớp này cho phép bạn khám phá vô số chi tiết nằm sau “lý lịch” của một assembly (thông tin phiên bản, thông tin văn hóa, v.v..).
<b>EventInfo</b>	Cầm giữ thông tin liên quan đến một tình huống nào đó.
<b>FieldInfo</b>	Cầm giữ thông tin liên quan đến một vùng mục tin nào đó.
<b>MemberInfo</b>	Lớp cơ bản trừu tượng này định nghĩa những hành xử thông dụng đối với các kiểu dữ liệu <b>EventInfo</b> , <b>FieldInfo</b> , <b>MethodInfo</b> , và <b>PropertyInfo</b> .
<b>MethodInfo</b>	Cầm giữ thông tin liên quan đến một hàm hành sự nào đó.
<b>Module</b>	Cho phép bạn truy xuất một module nào đó trong lòng một multifile assembly.
<b>ParameterInfo</b>	Cầm giữ thông tin liên quan đến một thông số nào đó.
<b>PropertyInfo</b>	Cầm giữ thông tin liên quan đến một thuộc tính nào đó.

### 4.2.2.2 Sử dụng *System.Reflection.MemberInfo*

Bạn bắt đầu bằng cách khởi gán một đối tượng kiểu dữ liệu **MemberInfo**. **MemberInfo** được dẫn xuất từ lớp **Type** nên cho gói ghém thông tin liên quan đến các thành viên thuộc một lớp (nghĩa là hàm hành sự, thuộc tính, tình huống, vùng mục tin, v.v..) Đối tượng **MemberInfo** này, thuộc namespace **System.Reflection**, được cung cấp để phát hiện những thuộc tính của một thành viên và cung cấp truy xuất lên metadata:

```
System.Reflection.MemberInfo inf = typeof(MyMath);
```

Bạn triệu gọi tác tử **typeof** đối với kiểu dữ liệu **MyMath**, cho trả về một đối tượng kiểu dữ liệu **Type** được dẫn xuất từ **MemberInfo**.

Bước kế tiếp là triệu gọi hàm **GetCustomAttributes** đối với đối tượng **MemberInfo** này, trao qua kiểu dữ liệu của attribute mà bạn muốn tìm ra. Bạn sẽ nhận về là một bản dãy các đối tượng kiểu dữ liệu **BugFixAttribute**.

```
object[] attributes;  
attributes = inf.GetCustomAttributes(typeof(BugFixAttribute), false);
```

Bây giờ, bạn có thể rảo qua bản dãy này, in ra các thuộc tính của đối tượng **BugFixAttribute**. Thí dụ 4-05 thay thế lớp Tester khỏi thí dụ 4-02.

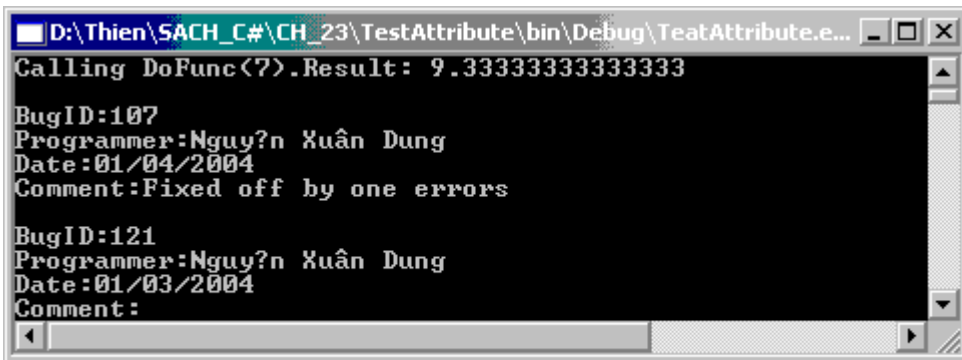
### Thí dụ 4-05: Sử dụng Reflection

```
public static void Main()  
{  
    MyMath mm = new MyMath();  
    Console.WriteLine("Calling DoFunc(7).Result: {0}", mm.DoFunc1(7));  
  
    // Đi lấy thông tin thành viên và rồi  
    // sử dụng thông tin này để tìm lại custom attribute  
    System.Reflection.MemberInfo inf = typeof(MyMath);  
    object[] attributes;  
    attributes = inf.GetCustomAttributes(typeof(BugFixAttribute),  
                                         false);  
  
    // rảo qua các attribute truy tìm các thuộc tính  
    foreach(Object attribute in attributes)  
    {  
        BugFixAttribute bfa = (BugFixAttribute) attribute;  
        Console.WriteLine("\nBugID:{0}", bfa.BugID);  
        Console.WriteLine("Programmer:{0}", bfa.ProgrammerName);  
        Console.WriteLine("Date:{0}", bfa.Date);  
        Console.WriteLine("Comment:{0}", bfa.Comment);  
    }  
}
```

Hình 4-08 cho thấy kết quả: metadata được in ra giống như bạn chờ đợi.

## 4.2.3 Phát hiện các kiểu dữ liệu

Bạn có thể dùng reflection để khảo sát và quan sát nội dung của một assembly. Bạn có thể tìm ra các kiểu dữ liệu được gắn liền với một module; các hàm hành sự, vùng mục tin, thuộc tính và tình huống được gắn liền với một kiểu dữ liệu, kể cả dấu ấn (signature) của mỗi hàm hành sự thuộc kiểu dữ liệu; các giao diện được hỗ trợ bởi kiểu dữ liệu; các lớp cơ bản trừu tượng của kiểu dữ liệu.



Hình 4-08: Kết xuất Thí dụ A-05, sử dụng Reflection

### 4.2.3.1 Nạp một assembly

Bước đầu tiên trong việc khảo sát nội dung của một .NET binary là nạp động assembly vào ký ức, sử dụng hàm hành sự static **Assembly.Load()**. Lớp Assembly gói ghém bản thân assembly hiện thời, dành cho mục đích reflection. Dấu ấn đối với hàm hành sự Load() là:

```
public static Assembly.Load(AssemblyName);
```

Giả sử bạn có một dự án console mang tên **CarReflector**, có một qui chiếu về assembly **CarLibrary** được tạo trong chương 3, “Tìm hiểu về Assembly và cơ chế Version”. Hàm hành sự static **Assembly.Load()** có thể giờ đây được triệu gọi bằng cách trao qua một chuỗi tên “CarLibrary”. Thí dụ 4-06 sau đây cho thấy cách nạp assembly CarLibrary:

#### Thí dụ 4-06: Lớp CarReflector

```
namespace CarReflector
{
    using System;
    using System.Reflection;
    using System.IO; // cần có để định nghĩa FileNotFoundException

    public class CarReflector
    {
        public static int Main(string[] args)
        {
            // sử dụng lớp Assembly để nạp CarLibrary
            Assembly a = null;
            try
            {
                a = Assembly.Load("CarLibrary");
```

```

    }
    catch (FileNotFoundException e)
    {
        Console.WriteLine(e.Message);
    }
    return 0;
}
}
}

```

Bạn để ý là hàm hành sự static **Assembly.Load()** được trao qua một tên thân thiện của assembly mà chúng ta quan tâm nạp vào ký ức. Như bạn có thể đoán trước, hàm này được nạp chồng (overloaded) nhiều lần để có thể cho phép bạn gắn kết theo nhiều cách khác nhau với một assembly. Thí dụ, bạn có thể đưa vào những chuỗi con ngoài cái tên thân thiện, chẳng hạn số phiên bản, trị public key, locale và strong name.

Nói một cách khác, tập hợp các item nhận diện một assembly được gọi là “display name” (tên dùng hiển thị). Dạng thức của một display name là một chuỗi phân cách bởi dấu phẩy bắt đầu bởi tên thân thiện, theo sau là những qualifier tùy chọn (có thể xuất hiện bất cứ thứ tự nào). Sau đây là khuôn mẫu bạn có thể theo. Các tiem tùy chọn được ghi trong dấu ngoặc:

```

Name (, Culture = CultureInfo) (, Ver = Major.Minor.Build.Revision)
(, SN = StrongName) (, PK = PublicKeyToken)

```

Khi cung cấp một display name, qui ước **SN=null** cho biết gắn kết và so khớp với một assembly được đặt tên đơn giản là bắt buộc. Ngoài ra, qui ước **Culture=""** cho biết là so khớp với culture mặc nhiên. Thí dụ minh hoạ như sau:

```

// Một AssemblyName trợn vẹn đối với simply named assembly
// với locale mặc nhiên
a = Assembly.Load(@"CarLibrary, Ver=1.0.454.30142, SN=null, Loc="");

```

Ngoài ra, bạn nên để ý là namespace **System.Reflection** cung cấp cho bạn kiểu dữ liệu **AssemblyName**, cho phép bạn biểu diễn chuỗi thông tin kể trên trên một thể hiện đối tượng tiện lợi. Điển hình lớp **AssemblyName** được sử dụng phối hợp với **System.Version** (một loại vỏ bọc thiên đối tượng xung quanh phiên bản của assembly). Một khi bạn đã thiết lập display name, nó có thể được trao qua cho hàm hành sự **Assembly.Load** được nạp chồng:

```

// Display name thiên đối tượng
AssemblyName asmName;
asmName = new AssemblyName();
asmName.Name = "CarLibrary";
Version v = new Version("1.0.454.30142");
asmName.Version = v;

```

```
a = Assembly.Load(asmName);
```

#### 4.2.3.2 *Liệt kê các kiểu dữ liệu trên một assembly được qui chiếu*

Một khi assembly đã được nạp vào rồi, bạn có thể khám phá ra tên của mỗi kiểu dữ liệu mà assembly chứa đựng bằng cách sử dụng hàm hành sự **Assembly.GetTypes()**. Hàm này sẽ trả về một bản dãy các đối tượng **Type**. Đối tượng **Type** là cốt lõi của reflection. **Type** tượng trưng cho những khai báo kiểu dữ liệu: class, interface, array, value và enumeration:

```
Assembly a = Assembly.Load("CarLibrary");  
Type[] types = a.GetTypes();
```

Bạn có thể dùng một vòng lặp **foreach** để rảo qua bản dãy các kiểu dữ liệu như theo thí dụ 4-07. Hàm hỗ trợ (helper) **ListAllTypes()** sẽ làm việc này. Vì dự án dùng đến class **Type**, nên bạn phải sử dụng namespace **System.Reflection**.

#### *Thí dụ 4-07: Liệt kê các kiểu dữ liệu trên một assembly*

```
using System;  
using System.Reflection;  
using System.IO;  
  
namespace CarReflector  
{  
    class CarReflector  
    {  
        static int Main(string[] args)  
        {  
            // sử dụng lớp Assembly để nạp CarLibrary  
            Assembly a = null;  
  
            try  
            {  
                a = Assembly.Load("CarLibrary");  
            }  
  
            catch(FileNotFoundException e)  
            {  
                Console.WriteLine(e.Message);  
            }  
  
            ListAllTypes(a);  
            Console.ReadLine();  
            return 0;  
        }  
    }  
}
```

```
private static void ListAllTypes(Assembly a)
{
    Console.WriteLine("Listing all types in {0}", a.FullName);
    Type[] types = a.GetTypes();
    foreach (Type t in types)
    {
        Console.WriteLine("Type: {0}", t);
    }
}
}
```

Hình 4-09 cho thấy kết quả chạy thí dụ 4-07 trên.



Hình 4-09: Liệt kê các kiểu dữ liệu của một assembly

## 4.2.4 Phản chiếu trên một kiểu dữ liệu

Bạn cũng có thể phản chiếu lên một kiểu dữ liệu (nghĩa là một lớp) sử dụng hàm hành sự **GetType()** (đừng nhầm **GetTypes()**). Thí dụ, bạn muốn trích kiểu dữ liệu **MiniVan** trên assembly **CarLibrary**, bạn viết:

```
Type theType = a.GetType("CarLibrary.Minivan");
```

### 4.2.4.1 Liệt kê các thành viên của một lớp

Khi đã có kiểu dữ liệu, **theType**, bạn dùng hàm hành sự **GetMembers()** được định nghĩa bởi lớp **Type** để liệt kê ra tất cả các thành viên của một kiểu dữ liệu (hàm hành sự, thuộc tính và vùng mục tin) như theo thí dụ 4-08. **GetMembers()** trả về một bản dãy kiểu dữ liệu **MemberInfo**.

**Thí dụ 4-08: Liệt kê các thành viên của một lớp**

```
using System;
using System.Reflection;
```

```

namespace CarReflector
{
    class CarReflector
    {
        static int Main(string[] args)
        {
            // Sử dụng lớp Assembly để nạp CarLibrary
            Assembly a = null;
            try
            {
                a = Assembly.Load("CarLibrary");
            }
            catch(FileNotFoundException e)
            {
                Console.WriteLine(e.Message);
            }

            ListAllMembers(a);
            Console.ReadLine();
            return 0;
        }

        private static void ListAllMembers(Assembly a)
        {
            Console.WriteLine("\nListing all members for
                               CarLibrary.Minivan");
            Type theType = a.GetType("CarLibrary.Minivan");
            MemberInfo[] mi = theType.GetMembers();
            foreach(MemberInfo m in mi)
            {
                Console.WriteLine("Type {0}: {1} ",
                                   m.MemberType.ToString(), m);
            }
        }
    }
}

```

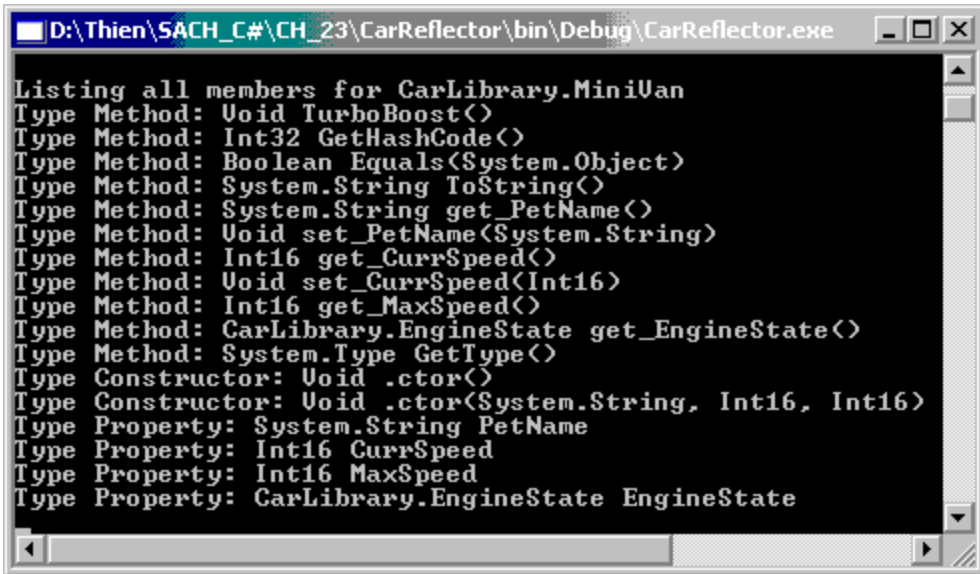
Hình 4-10 cho thấy kết quả chương trình:

#### 4.2.4.2 Truy tìm các thành viên đặc biệt của kiểu dữ liệu

Bạn có thể đi xa hơn bằng cách dùng hàm hành sự **FindMembers()** để tìm ra những thành viên đặc biệt nào đó của kiểu dữ liệu. Đây là một kiểu thanh lọc. Thí dụ, bạn có thể truy tìm các hàm hành sự theo tên bắt đầu bởi các mẫu tự “GET”.

Để thu hẹp lại cuộc truy tìm, bạn sử dụng hàm hành sự **FindMembers()**; hàm này trả về một bản dãy được lọc qua gồm các đối tượng kiểu **MemberInfo** đối với một thành viên thuộc kiểu dữ liệu được khai báo. Hàm **FindMembers()** nhận 4 thông số: **MemberTypes**, **BindingFlags**, **MemberFilter**, và **object**:



The screenshot shows a window titled 'D:\Thien\SACH\_C#\CH\_23\CarReflector\bin\Debug\CarReflector.exe'. The main content area displays a list of members for 'CarLibrary.MiniVan'. The list includes methods like TurboBoost, GetHashCode, Equals, ToString, get\_PetName, set\_PetName, get\_CurrSpeed, set\_CurrSpeed, get\_MaxSpeed, get\_EngineState, and GetType. It also lists constructors (.ctor) and properties (PetName, CurrSpeed, MaxSpeed, EngineState).

```
Listing all members for CarLibrary.MiniVan
Type Method: Void TurboBoost()
Type Method: Int32 GetHashCode()
Type Method: Boolean Equals(System.Object)
Type Method: System.String ToString()
Type Method: System.String get_PetName()
Type Method: Void set_PetName(System.String)
Type Method: Int16 get_CurrSpeed()
Type Method: Void set_CurrSpeed(Int16)
Type Method: Int16 get_MaxSpeed()
Type Method: CarLibrary.EngineState get_EngineState()
Type Method: System.Type GetType()
Type Constructor: Void .ctor()
Type Constructor: Void .ctor(System.String, Int16, Int16)
Type Property: System.String PetName
Type Property: Int16 CurrSpeed
Type Property: Int16 MaxSpeed
Type Property: CarLibrary.EngineState EngineState
```

Hình 4-10: Liệt kê các thành viên của lớp MiniVan

**MemberTypes:** đối tượng MemberTypes cho biết loại thành viên cần lọc tìm, bao gồm All, Constructor, Custom, Event, Field, Method, NestedType, Property, và TypeInfo.

- **BindingFlags:** Đây là một enumeration điều khiển việc lọc tìm. Có khá nhiều trị **BindingFlag**, bao gồm **IgnoreCase**, **Instance**, **Public**, **Static**, v.v.. Trị mặc nhiên cho biết không có binding flag, cho biết bạn không muốn hạn chế lọc tìm.
- **MemberFilter:** Một delegate dùng thanh lọc danh sách các thành viên trong bản dãy các đối tượng **MemberInfo**. Bộ sàng lọc mà bạn dùng đến là **Type.FilterName**, một vùng mục tin kiểu dữ liệu **Type** dùng thanh lọc đối với một tên.
- **Object:** Một trị kiểu chuỗi mà bộ lọc sẽ dùng đến. Trong trường hợp này, bạn sẽ trao "Get\*" để so khớp chỉ với những hàm hành sự nào bắt đầu bởi các mẫu tự Get.

Thí dụ 4-09 cho thấy hàm hành sự **GetParticularMember()** thêm vào dự án CarReflector để lọc tìm những thành viên đặc biệt.

**Thí dụ 4-09: Lọc tìm những thành viên đặc biệt**

```
private static void GetParticularMember(Assembly a)
{
    Console.WriteLine("\nFind particular members");
    Type theType = a.GetType("CarLibrary.MiniVan");
    MemberInfo[] mi = theType.FindMembers(MemberTypes.Method,
        BindingFlags.Instance |
        BindingFlags.Public |
        BindingFlags.Static, Type.FilterName, "Get*");

    foreach(MemberInfo m in mi)
    {
        Console.WriteLine("{0} is a {1}", m, m.MemberType.ToString());
    }
}
```

Hình 4-11 cho thấy kết quả



Hình 4-11: Liệt kê các thành viên của lớp

### 4.2.4.3 Liệt kê các hàm hành sự hoặc các thông số của một hàm hành sự

Bạn cũng có thể tập trung vào tất cả các hàm hành sự bằng cách dùng **GetMethods()**, **m lớp Type** hoặc chỉ một hàm hành sự đơn độc nào đó bằng cách sử dụng hàm hành sự **GetMethod** **m lớp Type**. Và khi có một hàm hành sự, bạn có thể sử dụng **GetParameters()** để khảo sát các thông số của hàm này. Để minh họa, giả sử bạn định nghĩa thêm hàm hành sự sau đây **TurnOnRadio()** trong dự án **CarLibrary** đối với lớp **Car**.

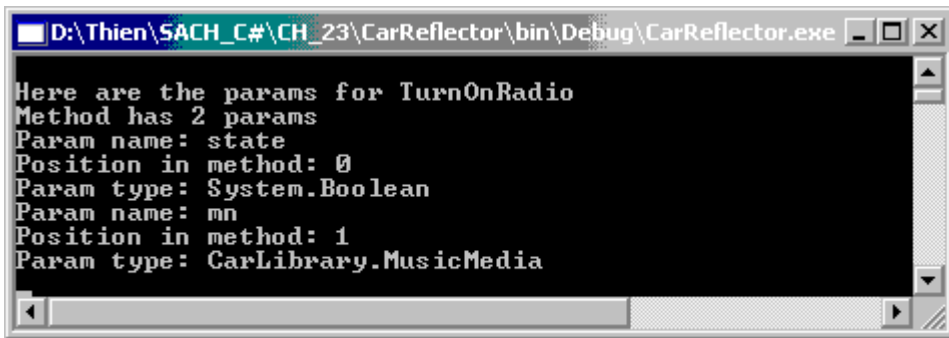
```
// Một lớp mới thêm vào lớp Car
public void TurnOnRadio(bool state, MusicMedia mn)
{
    if(state)
        MessageBox.Show("Jamming with {0}", mn.ToString());
    else
        MessageBox.Show("Quiet time...");
}
```

```
}
```

Hàm **TurnOnRadio()** tiếp nhận hai thông số, với thông số thứ hai là trị của enum **MusicMedia** được định nghĩa như sau:

```
public enum MusicMedia
{
    musicCD,
    musicTape,
    musicRadio
}
```

Bạn sử dụng hàm **MethodInfo.GetParameters()** để lấy ra thông tin liên quan đến các thông số. Hàm này trả về một bản dãy kiểu dữ liệu **ParameterInfo**. Mỗi item trong bản dãy này chứa vô số thuộc tính của một thông số. Thí dụ 4-09 sau đây là một hàm static khác thuộc lớp **CarReflector**, **GetParams()**, cho hiển thị các chi tiết khác nhau của mỗi thông số trên hàm hành sự **TurnOnRadio()**, như theo hình 4-12:



Hình 4-12: Thông tin các thông số của hàm **TurnOnRadio**

#### Thí dụ 4-10: Đi lấy thông tin trên hàm hành sự **TurnOnRadio()**

```
private static void GetParams(Assembly a)
{
    // Đi lấy kiểu dữ liệu MethodInfo
    Type theType = a.GetType("CarLibrary.MiniVan");
    MethodInfo mi = theType.GetMethod("TurnOnRadio");

    // Cho thấy số lượng thông số
    Console.WriteLine("\nHere are the params for {0}", mi.Name);
    ParameterInfo[] pi = mi.GetParameters();
    Console.WriteLine("Method has " + pi.Length + " params");

    // Cho thấy một vài thông tin đối với thông số
    foreach(ParameterInfo p in pi)
    {
        Console.WriteLine("Param name: {0}", p.Name);
        Console.WriteLine("Position in method: {0}", p.Position);
        Console.WriteLine("Param type: {0}", p.ParameterType);
    }
}
```

```
}
```

Hình 4-12 ở trên cho thấy kết xuất.

## 4.2.5 Tìm hiểu triệu gọi động trễ (late binding)

Một khi bạn đã phát hiện ra một hàm hành sự, bạn có khả năng triệu gọi hàm này bằng cách sử dụng reflection. Namespace **System.Reflection** còn cho phép bạn thực hiện việc kết nối trễ (late binding) đối với một kiểu dữ liệu, nghĩa là không triệu gọi hàm vào lúc biên dịch mà vào lúc chạy. Do đó gọi là triệu gọi động (dynamic invocation). Late binding là một kỹ thuật cho phép bạn giải quyết sự hiện hữu của một kiểu dữ liệu nào đó cùng với các thành viên của nó vào lúc chạy, thay vì vào lúc biên dịch. Một khi sự hiện hữu của một kiểu dữ liệu đã được xác lập, lúc ấy bạn có thể triệu gọi các hàm, truy xuất các thuộc tính và thao tác lên các vùng mục tin của một kiểu dữ liệu nào đó. Thí dụ, có thể bạn muốn triệu gọi hàm **TurboBoost()** của **CarLibrary.MiniVan** chẳng hạn.

Muốn triệu gọi hàm **TurboBoost()**, trước tiên bạn sẽ đi lấy thông tin **Type** đối với lớp **CarLibrary.MiniVan**:

```
Type miniVanType = Type.GetType("CarLibrary.MiniVan");
```

Với thông tin kiểu dữ liệu, bạn có thể nạp động một thể hiện của lớp này bằng cách dùng một hàm hành sự static của lớp **Activator**.

### 4.2.5.1 Lớp *Activator*

Lớp **System.Activator** là lớp chủ chốt trong việc kết nối trễ. Lớp này gồm 4 hàm hành sự static cho phép bạn tạo tại chỗ hoặc từ xa những đối tượng hoặc nhận về những qui chiếu đối với các đối tượng hiện hữu. Các hàm hành sự này là:

- **CreateComInstanceFrom**: Hàm này dùng tạo những thể hiện của các đối tượng COM.
- **CreateInstanceFrom**: Hàm này dùng tạo một qui chiếu về một đối tượng từ một assembly đặc biệt và tên kiểu dữ liệu.
- **GetObject**: Hàm này dùng marshal các đối tượng. Marshaling sẽ được đề cập đến tại chương 5, “Marshaling và Remoting”.
- **CreateInstance**: Hàm này dùng tạo những thể hiện của một đối tượng tại chỗ hoặc từ xa. Hàm này có vô số nạp chồng cho phép tạo một cách uyển chuyển

những thể hiện khác nhau. Một trong biến tấu là **CreateInstance()** nhận một đối tượng **Type** hợp lệ.

Với thí dụ **CarLibrary** trên, bạn có thể dùng hàm **CreateInstance()** để thể hiện một đối tượng lớp **CarLibrary.MiniVan** viết như sau, sử dụng đến lớp **Activator**:

```
Type miniVanType = Type.GetType("CarLibrary.MiniVan");
Object theObj = Activator.CreateInstance(miniVanType);
```

Giờ đây bạn có trong tay hai đối tượng: một đối tượng **Type** mang tên **miniVanType** mà bạn đã tạo ra bằng cách triệu gọi hàm **GetType()**, và một thể hiện của lớp **CarLibrary.MiniVan** mang tên **theObj**, mà bạn cho hiển lộ bằng cách triệu gọi hàm **CreateInstance()**. Tới đây, biến **theObj** chỉ về một thể hiện **CarLibrary.MiniVan** trong ký ức được tạo gián tiếp sử dụng lớp **Activator**.

Trước khi bạn có thể triệu gọi một hàm hành sự đối với đối tượng, bạn phải đi lấy hàm hành sự bạn cần đến từ đối tượng **Type**, **miniVanType**. Muốn thế, bạn phải dùng hàm **Type.GetMethod()** để nhận một đối tượng **MethodInfo**. Từ đối tượng **MethodInfo**, bạn mới có khả năng triệu gọi hàm bằng cách dùng **Invoke()**. Hàm **MethodInfo.Invoke()** đòi hỏi bạn trao qua tất cả các thông số cần phải đưa cho hàm hành sự được tượng trưng bởi **MethodInfo**. Các thông số này được tượng trưng bởi một bản dãy các **Objects**. Vì **TurboBoost()** không đòi hỏi bất cứ thông số nào, bạn chỉ cần trao "null" (cho biết hàm hành sự này không có thông số) là đủ.

Thí dụ 4-10 minh hoạ việc kết nối trễ chúng tôi vừa phác hoạ:

#### **Thí dụ 4-10: Late binding**

```
using System;
using System.Reflection;
using System.IO;

namespace LateBinding
{
    class LateBinding
    {
        public static int Main(string[] args)
        {
            // Dùng lớp Assembly để nạp CarLibrary
            Assembly a = null;
            try
            {
                a = Assembly.Load("CarLibrary");
            }
            catch (FileNotFoundException e)
            {
                Console.WriteLine(e.Message);
            }
        }
    }
}
```

```
// Đi lấy MiniVan type
Type miniVan = a.GetType("CarLibrary.MiniVan");

// Tạo một MiniVan vào lúc chạy (on the fly)
Object theObj = Activator.CreateInstance(miniVan);

// Đi lấy info đối với TurboBoost
MethodInfo mi = miniVan.GetMethod("TurboBoost");

// Triệu gọi hàm TurboBoost
mi.Invoke(theObj, null);

return 0;
}
}
```

Bây giờ, giả sử bạn muốn triệu gọi hàm **TellChildToBeQuiet()**, một hàm mới được đưa vào lớp **MiniVan**, và được định nghĩa như sau:

```
// Đoàn quân im lặng một chút đi
public void TellChildToBeQuiet(string kidName, int shameIntensity)
{
    for(int i = 0; i < shameIntensity; i++)
        MessageBox.Show("Be quiet " + kidName);
}
```

Hàm **TellChildToBeQuiet()** nhận hai thông số. Trong trường hợp này, trước khi triệu gọi hàm **Invoke()**, bạn phải chuẩn bị bản dãy các thông số, **paramArray**, như sau:

```
// Bây giờ một hàm hành sự với thông số
object[] paramArray = new object[2];
paramArray[0] = "Thien"; // Child name
paramArray[1] = 4; // Shame Intensity
mi = miniVan.GetMethod("TellChildToBeQuiet");
mi.Invoke(theObj, paramArray);
```

Nếu bạn cho chạy chương trình này, bạn sẽ thấy trước tiên khung đối thoại thứ nhất hiện lên do việc triệu gọi hàm **TurboBoost()**, (hình 4-13A) sau đó khi bạn ấn OK lên khung đối thoại này thì một khung đối thoại thứ hai hiện lên do việc triệu gọi hàm **TellChildToBeQuiet()** được gọi vào (hình 4-13B). Với khung đối thoại thứ hai bạn phải ấn OK 4 bốn lần nó mới biến đi.



**Hình 4-13: Late binding đối với hàm TurboBoost() và TellChildToBeQuiet()**

Bạn thấy là chỉ triệu gọi một hàm thôi mà cũng tốn bao nhiêu công sức. Tuy nhiên, sức mạnh ở chỗ là bạn có thể sử dụng reflection để phát hiện một assembly trên máy người sử dụng, sử dụng

reflection để truy vấn xem những hàm hành sự nào có sẵn rồi sau đó cũng sử dụng reflection để triệu gọi một cách động một trong những thành viên này.

## 4.3 Tìm hiểu và Xây dựng Dynamic Assembly

Tới đây bạn đã biết reflection được dùng trong 3 mục đích: nhìn xem metadata, phát hiện kiểu dữ liệu và triệu gọi động hàm hành sự. Bạn có thể sử dụng các kỹ thuật trên khi xây dựng những công cụ (chẳng hạn một môi trường triển khai hệ thống) hoặc khi xử lý các kịch bản. Tuy nhiên, sử dụng cực mạnh của reflection là với **reflection emit**.

Ta phải phân biệt giữa static assembly và dynamic assembly. **Static assembly** là những gì ta đã tham khảo từ trước đến nay trong tập sách này. Nói một cách đơn giản, static assembly thường được nạp từ đĩa cứng, nghĩa là chúng nằm đâu đó trên đĩa cứng dưới dạng một tập tin vật lý (hoặc dưới dạng nhiều tập tin nếu là một multifile assembly).

Còn **dynamic assembly** thì sẽ được tạo một cách động trong ký ức vào lúc chạy (“on the fly”) dùng đến chức năng cung cấp bởi namespace **System.Reflection.Emit**. Namespace này cho phép tạo một assembly với các module cùng kiểu dữ liệu liên hệ vào lúc runtime. Một khi bạn đã tạo động một assembly như thế, bạn hoàn toàn tự do cho cất trữ động kiểu dữ liệu mới của bạn lên đĩa. Thế là kết quả bạn nhận được sẽ là một static assembly. Ngoài ra, sử dụng namespace **System.Reflection.Emit**, bạn có thể thêm một cách động những kiểu dữ liệu mới cũng như những thành viên vào biểu diễn runtime của một assembly hiện hữu.

### 4.3.1 Tìm hiểu namespace System.Reflection.Emit

Các kiểu dữ liệu được định nghĩa trên namespace **System.Reflection.Emit** sẽ rất hữu ích đối với những ai lo tạo những công cụ phần mềm hoặc triển khai những ngôn ngữ lập trình. Thí dụ, bạn thử tưởng tượng bạn được giao nhiệm vụ tạo một phiên bản QuickBasic nhằm chạy trên .NET runtime.

Sử dụng namespace **System.Reflection.Emit** bạn có thể lấy đoạn mã thô BASIC và emit ngôn ngữ IL .NET tương ứng để rồi sau đó cho trữ lên một assembly được tạo ra một cách động. Trong khi công việc này xem ra khó tin nổi, nhưng các ngôn ngữ “ăn ý” với .NET (chẳng hạn JScript .NET) sử dụng kỹ thuật giống như thế. Trước tiên, bảng 4-05 liệt kê một số kiểu dữ liệu cốt lõi được định nghĩa trong namespace **System.Reflection.Emit**.

*Bảng 4-05: Các thành viên cốt lõi của namespace System.Reflection.Emit*

Các kiểu dữ liệu	Mô tả
<b>AssemblyBuilder</b>	Dùng tạo vào lúc chạy một assembly. Kiểu dữ liệu này có thể được dùng để tạo cả DLL hoặc EXE binary assembly. Các EXE phải triệu gọi hàm <b>ModuleBuilder.SetEntry Point()</b> để đặt điểm khởi đầu nhập đối với module. Nếu không có entry point, thì một DLL sẽ được kết sinh.
<b>ModuleBuilder</b>	Dùng tạo vào lúc chạy một module trong lòng một assembly.
<b>EnumBuilder</b> <b>TypeBuilder</b>	Tạo vào lúc chạy một kiểu dữ liệu (class, interface, v.v..) trong lòng một module.
<b>MethodBuilder</b> <b>EventBuilder</b> <b>LocalBuilder</b> <b>PropertyBuilder</b> <b>FieldBuilder</b> <b>ConstructorBuilder</b> <b>CustomAttributeBuilder</b>	Các item này (và một số khác) dùng tạo vào lúc chạy một thành viên của một kiểu dữ liệu nào đó (các hàm hành sự, các biến cục bộ, thuộc tính, hàm constructor, attribute)
<b>ILGenerator</b>	Kiểu này được dùng vào lúc chạy để tạo ra ngôn ngữ trung gian (IL) nằm sau đối với một thành viên.

### 4.3.2 Emitting<sup>12</sup> một Dynamic Assembly

Mục tiêu của mục này là tạo một single file assembly chỉ gồm một module. Trong lòng module, một lớp được mang tên HelloWorld. Lớp này có một hàm constructor (nhận về một thông số) dùng gán một trị cho một biến thành viên private (Msg) kiểu chuỗi. Ngoài ra, ta thử cho hỗ trợ một hàm hành sự public mang tên SayHello(), in ra một lời chúc mừng kiểu standard IO stream, và một hàm hành sự khác mang tên GetMsg() trả về một chuỗi nội tại private. Thật ra, bạn sẽ xây dựng thông qua lập trình lớp sau đây:

```
// Lớp này sẽ được xây dựng
// vào lúc chạy sử dụng System.Reflection.Emit
public class HelloWorld
{
    private string Msg;

    // giao diện public đối với lớp
    HelloWorld(string s) { Msg = s; }
    public string GetMsg() { return Msg; }
    public void SayHello() { System.Console.WriteLine("Hello there!"); }
}
```

<sup>12</sup> Emit, emitting: tạm thời chúng tôi không dịch từ này, vì dịch nghe ra nó vô duyên thế nào ấy.



Giả sử bạn tạo một dự án Console Application cho mang tên **DynAsmBuilder**. Lớp đầu tiên trong lồng dự án (MyAsmBuilder) chỉ có một hàm hành sự (CreateMyAsm) lo việc xây dựng một dynamic assembly, thiết lập một HelloClass, và cho cất trữ kiểu dữ liệu lên đĩa. Thí dụ 4-11 sau đây cho thấy toàn bộ đoạn mã lo công việc vừa kể trên:

#### ***Thí dụ 4-11: Dự án DynAsmBuilder***

```
using System;
using System.Reflection.Emit;
using System.Reflection;
using System.Threading;

namespace DynAsmBuilder
{
    class MyAsmBuilder
    {
        // Phía triệu gọi phát đi một AppDomain type
        public int CreateMyAsm(AppDomain curAppDomain)
        {
            // Tạo một cái tên cho assembly
            AssemblyName assemblyName = new AssemblyName();
            assemblyName.Name = "MyAssembly";
            assemblyName.Version = new Version("1.0.0.0");

            // Tạo assembly trong ký ức
            AssemblyBuilder assembly =
                curAppDomain.DefineDynamicAssembly(assemblyName,
                    AssemblyBuilderAccess.Save);

            // Đây là single file assembly nên tên module
            // trùng với tên assembly
            ModuleBuilder module = assembly.DefineDynamicModule(
                "MyAssembly", "MyAssembly.dll");

            // Cho định nghĩa một lớp public mang tên "HelloWorld"
            TypeBuilder helloWorldClass = module.DefineType(
                "MyAssembly.HelloWorld", TypeAttributes.Public);

            // Định nghĩa một private string Msg
            FieldBuilder msgField = helloWorldClass.DefineField(
                "Msg", Type.GetType("System.String"),
                FieldAttributes.Private);

            // Tạo hàm constructor HelloWorld(String s)
            Type[] constructorArgs = new Type[1];
            constructorArgs[0] = Type.GetType("System.String");
            ConstructorBuilder constructor =
                helloWorldClass.DefineConstructor(
                    MethodAttributes.Public,
                    CallingConventions.Standard, constructorArgs);
            ILGenerator constructorIL = constructor.GetILGenerator();
            constructorIL.Emit(OpCodes.Ldarg_0);
```

```

        Type objectClass = Type.GetType("System.Object");
        ConstructorInfo superConstructor =
            objectClass.GetConstructor(new Type[0]);
        constructorIL.Emit(OpCodes.Call, superConstructor);
        constructorIL.Emit(OpCodes.Ldarg_0);
        constructorIL.Emit(OpCodes.Ldarg_1);
        constructorIL.Emit(OpCodes.Stfld, msgField);
        constructorIL.Emit(OpCodes.Ret);

        // Bây giờ tạo hàm public string GetMsg
        MethodBuilder getMsgMethod = helloWorldClass.DefineMethod(
            "GetMsg", MethodAttributes.Public,
            Type.GetType("System.String"), null);
        ILGenerator methodIL = getMsgMethod.GetILGenerator();
        methodIL.Emit(OpCodes.Ldarg_0);
        methodIL.Emit(OpCodes.Ldfld, msgField);
        methodIL.Emit(OpCodes.Ret);

        // Tạo hàm public void SayHello
        MethodBuilder sayHiMethod = helloWorldClass.DefineMethod(
            "SayHello", MethodAttributes.Public, null, null);
        methodIL = sayHiMethod.GetILGenerator();
        methodIL.EmitWriteLine("Hello there!");
        methodIL.Emit(OpCodes.Ret);

        // "Nướng" lớp HelloWorld
        helloWorldClass.CreateType();

        // Cho cất trữ assembly lên đĩa
        assembly.Save("MyAssembly.dll");
        return 0;
    }

    static void Main(string[] args)
    {
    }
}

```

Thân hàm **CreateMyAsm()** bắt đầu thiết lập một tập hợp tối thiểu những đặc tính liên quan đến assembly của bạn, sử dụng lớp **AssemblyName**. Tiếp theo, ta tạo assembly trong ký ức sử dụng kiểu dữ liệu **AppDomain**, được mô tả ở chương 6, “Mạch trình và Đồng bộ hoá”.

```

// Tạo assembly trong ký ức
AssemblyBuilder assembly =
    curAppDomain.DefineDynamicAssembly(assemblyName,
    AssemblyBuilderAccess.Save);

```

Khi triệu gọi hàm **AppDomain.DefineDynamicAssembly()**, bạn phải khai báo chế độ truy xuất dựa theo những trị được ghi trong bảng 4-06.

**Bảng 4-06: Trị của Enumeration *AssemblyBuilderAccess***

Trị Enumeration	Mô tả
<b>Run</b>	Cho biết một dynamic assembly có thể được thi hành nhưng không được cất trữ.
<b>RunAndSave</b>	Cho biết một dynamic assembly có thể được thi hành và được cất trữ.
<b>Save</b>	Cho biết một dynamic assembly có thể được cất trữ nhưng không được thi hành

Khi đã có đối tượng **assembly**, việc kế tiếp là cho chèn module vào assembly. Bạn nhớ cho hiện ta đang xây dựng một single file assembly nên chỉ có một module. Nếu bạn phải xây dựng một multi-file assembly sử dụng hàm hành sự **DefineDynamicModule()**, bạn có thể khai báo một thông số tùy chọn thứ hai, tượng trưng cho tên của một module nào đó (chẳng hạn myMod.dll). Khi bạn muốn tạo một single-file assembly, tên module (và tập tin binary) sẽ giống với tên assembly:

```
// Đây là single file assembly nên tên module
// trùng với tên assembly
ModuleBuilder module = assembly.DefineDynamicModule(
    "MyAssembly", "MyAssembly.dll");
```

Bây giờ, ta thật sự nhập cuộc. Bằng cách sử dụng hàm **ModuleBuilder.DefineType()**, bạn có khả năng đưa vào module một lớp, một structure hoặc giao diện rồi nhận về lại một qui chiếu **TypeBuilder** tượng trưng cho item mới tạo (trong trường hợp này là một lớp mang tên HelloWorld). Tới đây, bạn có thể chèn vào một biến thành viên **private string Msg**:

```
// Cho định nghĩa một lớp public mang tên "HelloWorld"
TypeBuilder helloWorldClass = module.DefineType(
    "MyAssembly.HelloWorld", TypeAttributes.Public);

// Định nghĩa một private string Msg
FieldBuilder msgField = helloWorldClass.DefineField(
    "Msg", Type.GetType("System.String"),
    FieldAttributes.Private);
```

Khi đến phiên tạo hàm constructor của lớp này, bạn cần đưa vào đoạn mã IL thô vào thân hàm constructor, chịu trách nhiệm gán những thông số nhập vào chuỗi nội tại private. Hàm hành sự **Emit()** của lớp **ILGenerator** sẽ lo việc đưa IL vào thi công các thành viên lớp. Bản thân **Emit()** thường xuyên sử dụng enumeration **OpCodes** để đưa IL vào. Thí dụ, **OpCodes.Ret** cho biết trị trả về của một triệu gọi hàm. **OpCodes.Stfld** lo gán trị cho một biến thành viên. Sau đây là phần logic xây dựng hàm constructor:

```
// Tạo hàm constructor HelloWorld(String s)
Type[] constructorArgs = new Type[1];
constructorArgs[0] = Type.GetType("System.String");
ConstructorBuilder constructor = helloWorldClass.DefineConstructor(
    MethodAttributes.Public,
    CallingConventions.Standard, constructorArgs);
ILGenerator constructorIL = constructor.GetILGenerator();
constructorIL.Emit(OpCodes.Ldarg_0);
Type objectClass = Type.GetType("System.Object");
ConstructorInfo superConstructor = objectClass.GetConstructor(new
    Type[0]);
// triệu gọi hàm constructor base class
constructorIL.Emit(OpCodes.Call, superConstructor);

// Nạp con trỏ 'this' của đối tượng lên stack
constructorIL.Emit(OpCodes.Ldarg_0);

// Nạp một hằng 4 bytes trị 0 lên virtual stack
constructorIL.Emit(OpCodes.Ldarg_1);
constructorIL.Emit(OpCodes.Stfld, msgField); // gán msgField
constructorIL.Emit(OpCodes.Ret); // return
```

Bây giờ thử quan sát hàm hành sự SayHello():

```
// Tạo hàm public void SayHello
MethodBuilder sayHiMethod = helloWorldClass.DefineMethod(
    "SayHello", MethodAttributes.Public, null, null);
methodIL = sayHiMethod.GetILGenerator();

// Viết một hàng lên console
methodIL.EmitWriteLine("Hello there!");
methodIL.Emit(OpCodes.Ret);
```

Ở đây, bạn đã thiết lập một hàm hành sự public (MethodAttributes.Public) không nhận vào thông số và không trả về gì cả. Bạn để ý triệu gọi hàm EmitWriteLine(). Hàm này thuộc lớp ILGenerator tự động viết một hàng lên standard output.

Tới đây xem như bạn đã hiểu qua vấn đề.

### 4.3.3 Sử dụng một Dynamic Assembly được kết sinh thế nào

Một khi bạn đã có phần lô gic lo tạo và cất trữ assembly, MyAssembly.dll, bạn chỉ cần một phần mềm lo đọc lôgic này. Trong Main() bạn tạo một AppDomain để chuyển vào hàm hành sự **CreateMyAsm()**. Một khi triệu gọi hàm này trở về bạn có thể thực hiện late binding để nạp assembly này vào ký ức, và triệu gọi các hàm của HelloWorld. Trong

đoạn mã sau đây, thí dụ 4-12, bạn sẽ thấy bạn có thể triệu gọi thể nào hàm constructor nạp chồng với một đối mục được khai báo cũng như chặn bắt trị trả về của GetMsg().

**Thí dụ 4-12: Đọc lại assembly được tạo động**

```
using System;
using System.Reflection.Emit;
using System.Reflection;
using System.Threading;
namespace DynAsmBuilder
{
    class MyAsmBuilder
    {
        public int CreateMyAsm(AppDomain curAppDomain)
        {
            . . .
            return 0;
        }

        public static int Main(string[] args)
        {
            // Đi lấy application domain hiện hành
            AppDomain currAppDomain = Thread.GetDomain();

            // Tạo một dynamic assembly
            MyAsmBuilder asmBuilder = new MyAsmBuilder();
            asmBuilder.CreateMyAsm(currAppDomain);

            // Nạp assembly vào
            Assembly a = Assembly.Load("MyAssembly");

            // Đi lấy kiểu dữ liệu HelloWorld
            Type hello = a.GetType("MyAssembly.HelloWorld");

            // Tạo đối tượng HelloWorld và gọi đúng hàm constructor
            object[] ctorArgs = new object[1];
            ctorArgs[0] = "My amazing message...";
            object theObj = Activator.CreateInstance(hello, ctorArgs);

            // Triệu gọi hàm SayHello và cho ra chuỗi được trả về
            MethodInfo mi = hello.GetMethod("SayHello");
            mi.Invoke(theObj, null);

            // Triệu gọi hàm GetMsg và in ra thông điệp
            mi = hello.GetMethod("GetMsg");
            Console.WriteLine(mi.Invoke(theObj, null));

            Console.ReadLine();
            return 0;
        }
    }
}
```

Bạn cho chạy chương trình và hình 4-14 cho thấy phần kết xuất và hình 4-15 cho thấy ILDasm.exe hiển thị MyAssembly.dll thế nào.

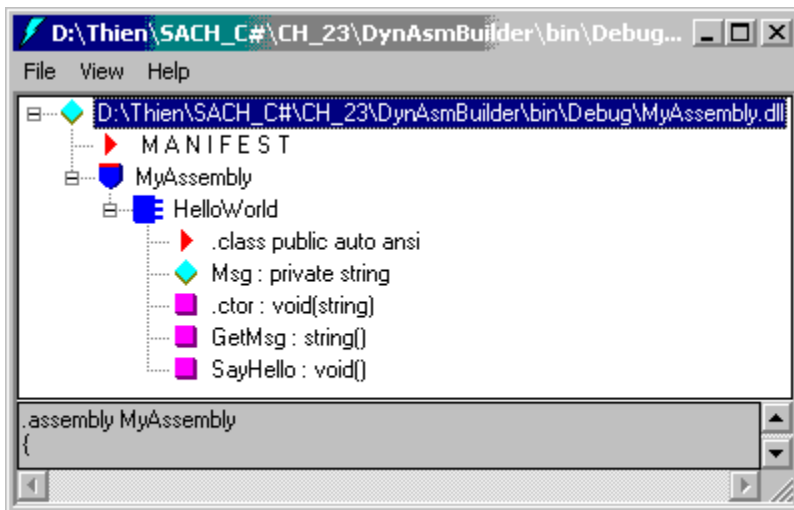


Hình 4-14: Triệu gọi các thành viên của assembly được tạo động

## 4.3.4 Reflection Emit

Như bạn thấy trong thí dụ đi trước, reflection emit hỗ trợ việc tạo động những kiểu dữ liệu mới vào lúc chạy. Bạn có thể định nghĩa một assembly để chạy động hoặc cho cất trữ lên đĩa, và bạn có thể định nghĩa những module cũng như những kiểu dữ liệu mới mà bạn có thể triệu gọi sau đó.

Để hiểu sâu sắc sức mạnh của reflection emit, bạn phải thử khảo sát một thí dụ hơi phức tạp thuộc dynamic invocation (triệu gọi động).



Hình 4-15: MyAssembly.dll do ILDasm.exe hiển thị

Có những vấn đề được giải quyết một cách tổng quát và thường chạy tương đối chậm, trong khi những giải pháp đặc thù thì lại chạy nhanh hơn. Để cho đơn giản vấn đề, ta thử xét đến hàm hành sự **DoSum()**, cho ra tổng của một chuỗi số nguyên từ **1...n**, theo đây người sử dụng cung cấp con số **n**.

Do đó, **DoSum(3)** sẽ bằng  $1+2+3$ , hoặc bằng 6. **DoSum(10)** sẽ bằng 55. Viết hàm này theo C# không khó chỉ mấy:

```
public int DoSum1(int n)
{
    int result = 0;
    for (int i = 0; i <= n; i++)
    {
        result += i;
    }
    return result;
}
```

Hàm trên đơn giản dùng một vòng lặp. Nếu bạn khở vào 3, thì hàm sẽ cộng  $1+2+3$  để cho ra 6. Với một con số lớn, và khi chạy nhiều lần như thế thì có thể sẽ chậm lại. Nếu ta cho con số 20, thì hàm này sẽ chạy nhanh hơn nếu ta cắt bỏ vòng lặp và viết như sau:

```
public int DoSum2()
{
    return = 1+2+3+4+5+6+7+8+9+10+11+12+13+14+15+16+17+18+19+20;
}
```

Hàm **DoSum2** chạy nhanh **DoSum1**. **DoSum2** thi hành theo “kiểu cụ trâu” (brute force). Nhanh bao nhiêu, bạn cần đặt một cái timer đo thời gian trên hai hàm này. Muốn thế, bạn sẽ dùng một đối tượng **DateTime** để đánh dấu thời gian khởi sự và một đối tượng **TimeSpan** để đo thời gian trải qua.

Đối với trắc nghiệm này, bạn cần tạo hai hàm hành sự **DoSum()**: hàm thứ nhất sẽ sử dụng vòng lặp còn hàm thứ hai thì không. Bạn cho triệu gọi mỗi hàm một triệu lần. Rồi sau đó bạn cho so sánh. Thí dụ, 4-13 minh họa toàn bộ chương trình trắc nghiệm.

### ***Thí dụ 4-13: So sánh Loop với Brute Force***

```
using System;
using System.Diagnostics;
using System.Threading;
namespace TestDoSum
{
    public class MyMath
    {
        public int DoSum(int n)
        {
            int result = 0;
            for(int i =0; i <= n; i++)
            { result += i; }
            return result;
        }

        public int DoSum2()
        {
```

```
        return 1+2+3+4+5+6+7+8+9+10+11+12+13+14+15+16+17+18+19+20;
    }
}

public class TestDriver
{
    static void Main()
    {
        const int val = 20; // val tổng cộng
        const int iterations = 1000000; // số lần lặp lại
        int result = 0; // kết quả

        MyMath m = new MyMath();

        DateTime startTime=DateTime.Now; // ghi thời gian khởi đi

        // trắc nghiệm đầu tiên theo vòng lặp
        for(int i=0; i<iterations; i++)
        {
            result = m.DoSum(val);
        }

        // tính thời gian trải qua và in ra kết quả lên console
        TimeSpan elapsed = DateTime.Now - startTime;
        Console.WriteLine("Loop: Sum of ({0}) = {1}", val, result);
        Console.WriteLine("The elapsed time in milliseconds is: " +
            elapsed.TotalMilliseconds.ToString());

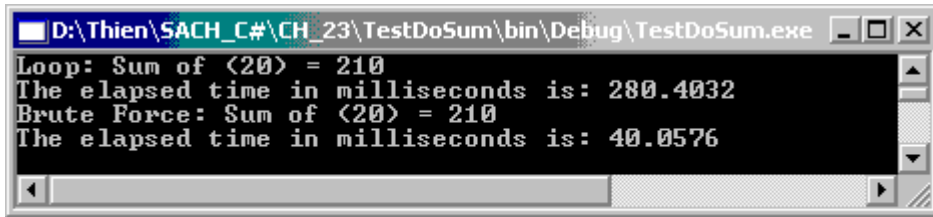
        // trắc nghiệm theo kiểu "cụ trâu" (brute force)
        startTime = DateTime.Now;
        for (int i=0; i<iterations; i++)
        {
            result = m.DoSum2();
        }

        // tính thời gian trải qua và in ra kết quả lên console
        elapsed = DateTime.Now - startTime;
        Console.WriteLine("Brute Force: Sum of ({0}) = {1}",
            val, result);
        Console.WriteLine("The elapsed time in milliseconds is: " +
            elapsed.TotalMilliseconds.ToString());

        Console.ReadLine();
    }
}
```

Hình 4-16 cho thấy kết quả thi hành chương trình trên:





```
D:\Thien\SACH_C#\CH_23\TestDoSum\bin\Debug\TestDoSum.exe
Loop: Sum of <20> = 210
The elapsed time in milliseconds is: 280.4032
Brute Force: Sum of <20> = 210
The elapsed time in milliseconds is: 40.0576
```

Hình 4-16: Kết xuất Test DoSum()

Như bạn có thể thấy cả hai hàm hành sự trả về cùng kết quả nhưng thời gian thi hành đối với trắc nghiệm kiểu “cụ trâu” (brute force) nhanh hơn kiểu vòng lặp đến 7 lần.

Có cách gì tránh sử dụng vòng lặp nhưng vẫn cung cấp một giải pháp tổng quát? Theo lập trình cổ điển, câu trả lời là không, nhưng với reflection thì bạn có một lựa chọn. Bạn có thể vào lúc chạy lấy trị người sử dụng muốn (trong trường hợp này là 20) rồi viết lên đĩa một lớp thi công giải pháp “cụ trâu”, rồi sau đó bạn có thể sử dụng dynamic invocation triệu gọi hàm này.

Ít nhất có 3 cách để thực hiện đi đến kết quả trên, mỗi cách tăng sự hấp dẫn. Cách thứ ba, reflection emit, là tốt nhất, nhưng xem kỹ hai cách kia cũng rất bổ ích cho việc học hỏi.

#### 4.3.4.1 Dynamic Invocation sử dụng đến InvokeMember()

Tiếp cận đầu tiên sẽ là tạo một lớp mang tên **BruteForceSums** một cách động, vào lúc chạy. Lớp này sẽ chứa một hàm hành sự, **ComputeSum()**, cho thi công cách tiếp cận kiểu “cụ trâu” này. Bạn sẽ viết lớp này lên đĩa, cho biên dịch nó rồi sau đó dùng dynamic invocation để triệu gọi hàm “cụ trâu” thông qua hàm **InvokeMember()** thuộc lớp **Type**. Điểm then chốt là BruteForceSum.cs không hiện diện cho tới khi bạn chạy chương trình. Bạn sẽ tạo ra nó khi thấy cần và cung cấp cho nó những đối mục sau đó.

Muốn thực hiện điều này, bạn sẽ phải tạo một lớp mới mang tên **ReflectionTest**. Công việc của lớp này là tạo động lớp **BruteForceSums**, viết lên đĩa và cho biên dịch lớp này. Lớp **ReflectionTest** chỉ có hai hàm hành sự **DoSum()** và **GenerateCode()**.

Hàm **ReflectionTest.DoSum()** là một hàm public trả về tổng cộng, khi ta cung cấp một trị **n**. Nghĩa là khi bạn trao qua  $n = 10$ , thì tổng cộng của  $1+2+3+4+5+6+7+8+9+10$ . Nó làm việc này bằng cách tạo ra một lớp **BruteForceSums** và ủy thác công việc cho hàm **ComputeSum**.

Lớp ReflectionTest có hai vùng mục tin private:

```
Type theType = null;  
object theClass = null;
```

Vùng mục tin đầu tiên là một đối tượng kiểu dữ liệu **Type**, mà bạn sẽ dùng để nạp lớp của bạn từ đĩa xuống. Còn vùng mục tin thứ hai là một đối tượng kiểu dữ liệu **object**, mà bạn sẽ dùng triệu gọi động hàm **ComputeSums()** của lớp **BruteForceSums** mà bạn sẽ tạo ra.

Chương trình **TestDriver** sẽ hiển lộ một thể hiện của lớp **ReflectionTest** và triệu gọi hàm **DoSum** của nó, trao qua trị. Đối với phiên bản của chương trình này, trị tăng lên 200.

Hàm **DoSum()** kiểm tra liệu xem **theType** có null hay không. Nếu là null, nghĩa là lớp **BruteForceSums** chưa được tạo động. **DoSum** sẽ triệu gọi hàm phụ trợ **GenerateCode** để kết sinh đoạn mã đối với lớp cũng như đối với hàm hành sự **ComputeSums**. Sau đó **GenerateCode** sẽ viết đoạn mã mới kết sinh lên đĩa dưới dạng một tập tin .cs, cho chạy trình biên dịch biến nó thành một assembly trên đĩa. Một khi việc này hoàn tất, **DoSum** có thể triệu gọi hàm sử dụng reflection.

Một khi lớp **BruteForceSums** và hàm hành sự **ComputeSums** được tạo ra, bạn nạp assembly từ đĩa và gán thông tin kiểu dữ liệu vào **theType**, và **DoSum** có thể dùng để triệu gọi động hàm **ComputeSum** để nhận câu trả lời chính xác.

Bạn bắt đầu tạo một hằng đối với trị bạn muốn lấy tổng:

```
const int val = 200; // 1..200
```

Mỗi lần bạn tính tổng, sẽ là tổng cộng các trị từ 1 đến 200. Trước khi bạn tạo lớp động, bạn cần trở lui và tạo lại **MyMath**:

```
MyMath m = new MyMath();
```

Bạn thêm một hàm **DoSumLooping** giống như với thí dụ trước, dùng làm thước đo (benchmark) so sánh với kiểu “cụ trâu”:

```
public int DoSumLooping(int initialVal)  
{  
    int result = 0;  
    for(int i=0; i<=initialVal;i++)  
    {  
        result += i;  
    }  
    return result;  
}
```

Bây giờ thì bạn sẵn sàng tạo ra một lớp năng động đem so sánh với kiểu vòng lặp. Trước tiên, bạn cho hiển lộ một đối tượng kiểu **ReflectionTest** và triệu gọi hàm **DoSum()**

```
ReflectionTest t = new ReflectionTest();
result = t.DoSum(val);
```

**ReflectionTest.DoSum** kiểm tra xem vùng mục tin kiểu **Type**, **theType**, có null hay không. Nếu null, có nghĩa là bạn chưa tạo và biên dịch lớp **BruteForceSums**, và bạn phải làm ngay bây giờ:

```
if(theType == null)
{
    GenerateCode(theValue);
}
```

Hàm **GenerateCode** sẽ lấy trị **theValue** (trong trường hợp này là 200) như là thông số cho biết bao nhiêu trị phải cộng lại. **GenerateCode** bắt đầu tạo một tập tin lên đĩa. Chi tiết liên quan đến xuất nhập dữ liệu đã được đề cập đến ở chương 1, “Xuất nhập dữ liệu & Sản sinh hàng loạt đối tượng”. Trước tiên, triệu gọi hàm static **File.Open**, trao qua tên tập tin và một flag cho biết bạn muốn tạo một tập tin. **File.Open** sẽ trả về một đối tượng **Stream**.

```
string fileName = "BruteForceSums";
Stream s = File.Open(fileName + ".cs", FileMode.Create);
```

Một khi bạn có một đối tượng **Stream**, bạn có thể tạo một đối tượng **StreamWriter** để viết những hàng văn bản lên tập tin. Bạn bắt đầu tập tin mới với dòng chú giải:

```
StreamWriter writer = new StreamWriter(s);
writer.WriteLine("// Dynamically created BruteForceSums class");
```

Dòng lệnh trên sẽ viết ra dòng văn bản:

```
// Dynamically created BruteForceSums class
```

lên tập tin vừa mới được tạo ra, **BruteForceSums.cs**. Tiếp theo là viết ra phân khai báo lớp:

```
string className = "BruteForceSums";
writer.WriteLine("class {0}", className);
writer.WriteLine("{");
```

Trong lòng dấu ngoặc nhọn “{” bạn tạo hàm hành sự **ComputeSum**:

```
writer.WriteLine("\tpublic double ComputeSum()");
```

```
writer.WriteLine("\t{");
writer.WriteLine("\t// Brute force sum method");
writer.WriteLine("\t// For Value = {0}", theVal);
```

Bây giờ đã đến lúc viết ra các lệnh cộng. Khi bạn làm xong, bạn muốn tập tin chứa dòng như sau:

```
return 0+1+2+3+4+5+...+200;
```

Các lệnh sau đây làm việc này, sử dụng hàm Write():

```
writer.Write("\treturn 0");
for (int i=1; i <= theVal; i++)
{
    writer.Write("+ {0}", i);
}
```

Bạn để ý \t trên dòng lệnh Write đầu tiên cho biết là viết canh thụt (indentation) qua phải một nấc. Khi vòng lặp hoàn tất, bạn kết thúc lệnh return bởi dấu chấm phẩy và cho đóng lại hàm hành sự và lớp:

```
writer.WriteLine(";");
writer.WriteLine("\t");
writer.WriteLine("}");
```

Cuối cùng bạn cho đóng lại đối tượng streamWriter và tập tin:

```
writer.Close();
s.Close();
```

Khi hàm này chạy, tập tin BruteForceSums.cs sẽ viết lên đĩa, giống như sau:

```
// Dynamically created BruteForceSums class
class BruteForceSums
{
    public double ComputeSum()
    {
        // Brute force sum method
        // For Value = 200
        return 0+ 1+ 2+ 3+ 4+ 5+ 6+ 7+ 8+ 9+ 10+ 11+ 12+ 13+ 14+ 15+ 16+ 17+
18+ 19+ 20+ 21+ 22+ 23+ 24+ 25+ 26+ 27+ 28+ 29+ 30+ 31+ 32+ 33+ 34+ 35+
36+ 37+ 38+ 39+ 40+ 41+ 42+ 43+ 44+ 45+ 46+ 47+ 48+ 49+ 50+ 51+ 52+ 53+
54+ 55+ 56+ 57+ 58+ 59+ 60+ 61+ 62+ 63+ 64+ 65+ 66+ 67+ 68+ 69+ 70+ 71+
72+ 73+ 74+ 75+ 76+ 77+ 78+ 79+ 80+ 81+ 82+ 83+ 84+ 85+ 86+ 87+ 88+ 89+
90+ 91+ 92+ 93+ 94+ 95+ 96+ 97+ 98+ 99+ 100+ 101+ 102+ 103+ 104+ 105+
106+ 107+ 108+ 109+ 110+ 111+ 112+ 113+ 114+ 115+ 116+ 117+ 118+ 119+
120+ 121+ 122+ 123+ 124+ 125+ 126+ 127+ 128+ 129+ 130+ 131+ 132+ 133+
134+ 135+ 136+ 137+ 138+ 139+ 140+ 141+ 142+ 143+ 144+ 145+ 146+ 147+
148+ 149+ 150+ 151+ 152+ 153+ 154+ 155+ 156+ 157+ 158+ 159+ 160+ 161+
162+ 163+ 164+ 165+ 166+ 167+ 168+ 169+ 170+ 171+ 172+ 173+ 174+ 175+
```

```

176+ 177+ 178+ 179+ 180+ 181+ 182+ 183+ 184+ 185+ 186+ 187+ 188+ 189+
190+ 191+ 192+ 193+ 194+ 195+ 196+ 197+ 198+ 199+ 200;
}
}

```

Công việc còn lại là xây dựng tập tin và sau đó sử dụng hàm hành sự. Muốn tạo tập tin, bạn bắt đầu một process mới. Chương 6, “Mạch trình và Đồng bộ hoá”, sẽ đi vào chi tiết đối với process. Cách tốt nhất khởi động một process là với một cấu trúc **ProcessStartInfo** cầm giữ command line. Bạn cho hiển lộ một đối tượng **ProcessStartInfo** rồi cho đặt đề tên tập tin của nó về *cmd.exe*:

```

ProcessStartInfo psi = new ProcessStartInfo();
psi.FileName = "cmd.exe";

```

Bạn cần trao qua chuỗi mà bạn muốn triệu gọi ở command line. Thuộc tính **ProcessStartInfo.Arguments** khai báo cho biết những đối mục command-line cần sử dụng khi khởi động chương trình. Các đối mục sẽ là /c cho *cmd.exe* biết là thoát ra khi thi hành lệnh, rồi sau đó là lệnh đối với *cmd.exe*. Lệnh đối với *cmd.exe* là lệnh biên dịch:

```

string compileString = "/c csc /optimize+ ";
compileString += "/target:library ";
compileString += "{0}.cs > compile.out";

```

Chuỗi **compileString** sẽ triệu gọi C# compiler (*csc*), cho biết phải tối ưu hóa đoạn mã và xây dựng một tập tin DLL (*/target:library*). Bạn chuyển kết xuất của biên dịch lên một tập tin mang tên *compile.out* để bạn có thể xem xét nếu có sai lầm. Bạn phối hợp **compileString** với **fileName**, sử dụng hàm static *Format* của lớp *string* và gán chuỗi phối hợp cho **psi.Arguments**.

```

psi.Arguments = String.Format(compileString, fileName);

```

Tác dụng của tất cả các việc này là đặt đề thuộc tính **Arguments** của đối tượng **ProcessStartInfo psi** về:

```

/c csc /optimize+ /target:library BruteForceSums.cs > compile.out

```

Trước khi triệu gọi *cmd.exe*, bạn cho đặt đề thuộc tính **WindowStyle** của **psi** về **Minimized**, như vậy khi lệnh thi hành cửa sổ sẽ không nhấp nháy và cho off hiển thị của người sử dụng:

```

psi.WindowStyle = ProcessWindowStyle.Minimized;

```

Bây giờ bạn đã sẵn sàng bắt đầu thi hành *cmd.exe* và bạn sẽ chờ cho tới khi nó xong việc mới tiếp tục với phần còn lại của hàm **GenerateCode**.

```
Process proc = Process.Start(psi);
proc.WaitForExit();
```

Một khi tiến trình biên dịch xong, bạn có thể đi lấy assembly, và từ assembly bạn có thể lấy lớp bạn vừa tạo ra. Cuối cùng bạn yêu cầu lớp này đối với kiểu dữ liệu và gán cho biến **theType**:

```
Assembly a = Assembly.LoadFrom(fileName + ".dll");
theClass = a.CreateInstance(className);
theType = a.GetType(className);
```

Bây giờ, bạn có thể gỡ bỏ tập tin .cs bạn vừa kết sinh:

```
File.Delete(fileName + ".cs");
```

**theType** đã được điền kiểu dữ liệu và bạn sẵn sàng trở về **DoSum** để triệu gọi động hàm **ComputeSum()**. Đối tượng Type có một hàm hành sự **InvokeMember()** mà bạn có thể dùng để triệu gọi một thành viên của lớp được mô tả bởi đối tượng **Type**. Hàm này được nạp chồng; phiên bản bạn sử dụng đến nhận vào 5 thông số:

```
public object InvokeMember(
    string name,
    BindingFlags invkeAttr,
    Binder binder,
    object target,
    object[] args
);
```

với

- **name**: là tên hàm hành sự bạn muốn triệu gọi.
- **invokeAttr**: là một bit mask **BindingFlags** cho biết làm thế nào dò tìm đối tượng được tiến hành. Trong trường hợp này, bạn sẽ dùng flag **InvokeMethod** cho OR với flag **Default**. Đây là những flag chuẩn dùng triệu gọi động một hàm hành sự.
- **binder**: dùng hỗ trợ việc chuyển đổi kiểu dữ liệu. Bằng cách trao qua null, bạn muốn cho biết bạn muốn default binder.
- **target**: là đối tượng theo đây bạn muốn triệu gọi hàm hành sự. Trong trường hợp này bạn trao **theClass**, đúng là lớp bạn vừa tạo ra.
- **args**: là một bản dãy các đối mục cần trao qua cho hàm hành sự bạn đang triệu gọi.

Lệnh triệu gọi InvokeMember giống như sau:

```
object[] arguments = new object[0];
object retVal = theType.InvokeMember("ComputeSum",
    BindingFlags.Default | BindingFlags.InvokeMethod,
    null,
    theClass,
    arguments);
return (double) retVal;
```

Kết quả triệu gọi hàm này được gán cho một biến cục bộ retVal, được trả về như là số double đối với chương trình driver. Thí dụ 4-14 cho thấy toàn bộ chương trình:

#### ***Thí dụ 4-14: Dynamic Invocation với Type và InvokeMember()***

```
using System;
using System.Diagnostics;
using System.Reflection;
using System.IO;

namespace DynInvocation
{
    // Lớp này dùng làm thước đo đối với kiểu vòng lặp
    public class MyMath
    {
        // tính tổng dùng vòng lặp
        public int DoSumLooping(int initialVal)
        {
            int result = 0;
            for(int i=0; i<=initialVal;i++)
            {
                result += i;
            }
            return result;
        }
    }

    // Lớp này chịu trách nhiệm tạo lớp BruteForceSums, cho biên dịch
    // và triệu gọi hàm DoSum một cách động
    public class ReflectionTest
    {
        // Hàm này được triệu gọi bởi test driver
        public double DoSum(int theValue)
        {
            // nếu không có qui chiếu về lớp thì phải tạo ra
            if(theType == null)
            {
                GenerateCode(theValue);
            }

            // với qui chiếu về lớp mới được tạo động bạn có thể
```

```
// triệu gọi hàm ComputeSum
object[] arguments = new object[0];
object retVal = theType.InvokeMember("ComputeSum",
    BindingFlags.Default | BindingFlags.InvokeMethod,
    null, theClass, arguments);
return (double) retVal;
}

// Cho kết sinh đoạn mã và cho biên dịch
private void GenerateCode(int theVal)
{
    // cho mở tập tin để viết
    string fileName = "BruteForceSums";
    Stream s = File.Open(fileName + ".cs", FileMode.Create);
    StreamWriter writer = new StreamWriter(s);
    writer.WriteLine("// Dynamically created BruteForceSums
        class");

    // Tạo ra lớp BruteForceSums
    string className = "BruteForceSums";
    writer.WriteLine("class {0}", className);
    writer.WriteLine("{");

    // Tạo ra hàm ComputeSum
    writer.WriteLine("\tpublic double ComputeSum()");
    writer.WriteLine("\t{");
    writer.WriteLine("\t\t// Brute force sum method");
    writer.WriteLine("\t\t// For Value = {0}", theVal);

    // viết ra kiểu cộng cụ thể
    writer.WriteLine("\treturn 0");
    for (int i=1; i <= theVal; i++)
    {
        writer.WriteLine("\t\t+ {0}", i);
    }
    writer.WriteLine("\t"); // kết thúc hàm
    writer.WriteLine("\t"); // end method
    writer.WriteLine("}"); // end class

    writer.Close(); // đóng lại writer
    s.Close(); // đóng lại stream

    // Xây dựng tập tin
    ProcessStartInfo psi = new ProcessStartInfo();
    psi.FileName = "cmd.exe";
    string compileString = "/c csc /optimize+ ";
    compileString += "/target:library ";
    compileString += "{0}.cs > compile.out";

    psi.Arguments = String.Format(compileString, fileName);
    psi.WindowStyle = ProcessWindowStyle.Minimized;

    Process proc = Process.Start(psi);
```



```

        proc.WaitForExit(); // chờ ít nhất 2 giây

        Assembly a = Assembly.LoadFrom(fileName + ".dll");
        theClass = a.CreateInstance(className);
        theType = a.GetType(className);
    }

    Type theType = null;
    object theClass = null;
}

public class TestDriver
{
    [STAThread]
    static void Main()
    {
        const int val = 200; // 1..200
        const int iterations = 100000;
        double result = 0;

        // Cho chạy benchmark
        MyMath m = new MyMath();
        DateTime startTime = DateTime.Now;

        for(int i=0; i<iterations; i++)
        {
            result = m.DoSumLooping(val);
        }
        TimeSpan elapsed = DateTime.Now - startTime;

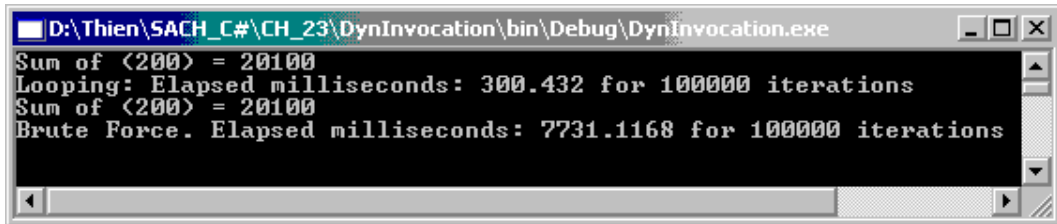
        Console.WriteLine("Sum of ({0}) = {1}", val, result);
        Console.WriteLine("Looping: Elapsed milliseconds: " +
            elapsed.TotalMilliseconds + " for {0} iterations",
            iterations);

        // Cho chạy kiểu Reflection
        ReflectionTest t = new ReflectionTest();
        startTime = DateTime.Now;
        for (int i=0; i<iterations; i++)
        {
            result = t.DoSum(val);
        }
        elapsed = DateTime.Now - startTime;
        Console.WriteLine("Sum of ({0}) = {1}", val, result);
        Console.WriteLine("Brute Force. Elapsed milliseconds: " +
            elapsed.TotalMillisecond + " for {0} iterations",
            iterations);

        Console.ReadLine();
    }
}

```

Hình 4-17 cho thấy kết xuất.



Hình 4-17: Kết xuất của thí dụ 4-14

Bạn thấy lần này, hàm được triệu gọi động chạy rất chậm (gần 28 lần) so với kiểu vòng lặp. Điều này không ngạc nhiên chỉ lắm vì viết tập tin lên đĩa, cho biên dịch, rồi đọc lại từ đĩa và triệu gọi hàm tất cả các điều tổn hao thời gian. Bạn đã đạt mục tiêu đề ra nhưng chiến thắng có vẻ ê chề.

#### 4.3.4.2 *Dynamic Invocation sử dụng Interfaces*

Xem ra là dyn invocation đặc biệt rất chậm. Bạn muốn duy trì cách tiếp cận tổng quát là viết ra lớp vào lúc chạy và biên dịch on the fly. Nhưng thay vì sử dụng dyn invocation bạn lại chỉ cần triệu gọi hàm hành sự mà thôi. Một cách để tăng tốc là sử dụng một giao diện để gọi trực tiếp hàm hành sự `ComputeSums()`.

Muốn thế, bạn cần thay đổi **ReflectionTest.DoSum()** từ đoạn mã:

```
public double DoSum(int theValue)
{
    // nếu không có qui chiếu về lớp thì phải tạo ra
    if(theType == null)
    {
        GenerateCode(theValue);
    }

    // với qui chiếu về lớp mới được tạo động bạn có thể
    // triệu gọi hàm ComputeSum
    object[] arguments = new object[0];
    object retVal = theType.InvokeMember("ComputeSum",
        BindingFlags.Default | BindingFlags.InvokeMethod,
        null, theClass, arguments);
    return (double) retVal;
}
```

qua đoạn mã sau đây:

```

public double DoSum(int theValue)
{
    if (theComputer == null)
    {
        GenerateCode(theValue);
    }
    return (theComputer.ComputeSum());
}

```

Trong thí dụ này, **theComputer** là một giao diện đối với một đối tượng kiểu dữ liệu **BruteForceSums**. Phải là một giao diện chứ không phải là một đối tượng vì khi bạn biên dịch chương trình này, theComputer chưa hiện hữu; bạn sẽ tạo nó một cách động. Bạn cho gỡ bỏ những khai báo đối với theType và theFunction và cho thay thế bởi:

```

IComputer theComputer = null;

```

Khai báo trên cho biết có một giao diện **IComputer**. Vào đầu chương trình bạn phải khai báo giao diện này như sau:

```

public interface IComputer
{
    double ComputeSum();
}

```

Khi bạn tạo lớp **BruteForceSums**, bạn phải bảo lớp này thi công **IComputer**:

```

writer.WriteLine("class {0}: DynInvocation.IComputer ", className);

```

Bạn cho cất trữ chương trình lên một tập tin dự án mang tên Reflection và thay đổi compileString trong GenerateCode như sau:

```

string compileString = "/c csc /optimize+ ";
compileString += "/r:\"DynInvocation.exe\"";
compileString += "/target:library ";
compileString += "{0}.cs > compile.out";

```

**CompileString** sẽ cần qui chiếu về bản thân chương trình DynInvocation (Reference.exe) như vậy trình biên dịch được triệu gọi động sẽ biết tìm ở đâu khai báo của **IComputer**:

```

theComputer = (IComputer)a.CreateInstance(className);

```

Bạn dùng giao diện để triệu gọi trực tiếp hàm trong DoSum.

```

return (theComputer.ComputeSum());

```

Thí dụ 4-15 là toàn bộ đoạn mã:

***Thí dụ 4-15: Dynamic Invocation với Interfaces)***

```
using System;
using System.Diagnostics;
using System.Reflection;
using System.IO;

namespace DynInvocation
{
    // Lớp này dùng làm thước đo đối với kiểu vòng lặp
    public class MyMath
    {
        // tính tổng dùng vòng lặp
        public int DoSumLooping(int initialVal)
        {
            int result = 0;
            for(int i=0; i<=initialVal;i++)
            {
                result += i;
            }
            return result;
        }
    }

    public interface IComputer
    {
        double ComputeSum();
    }

    // Lớp này chịu trách nhiệm tạo lớp BruteForceSums, cho biên dịch
    // và triệu gọi hàm DoSum một cách động
    public class ReflectionTest
    {
        // Hàm này được triệu gọi bởi test driver
        public double DoSum(int theValue)
        {
            // nếu không có qui chiếu về lớp thì phải tạo ra
            if (theComputer == null)
            {
                GenerateCode(theValue);
            }
            return (theComputer.ComputeSum());
        }

        // Cho kết sinh đoạn mã và cho biên dịch
        private void GenerateCode(int theVal)
        {
            // cho mở tập tin để viết
            string fileName = "BruteForceSums";
            Stream s = File.Open(fileName + ".cs", FileMode.Create);
```

```

StreamWriter writer = new StreamWriter(s);
writer.WriteLine("// Dynamically created BruteForceSums
    class");
// Tạo ra lớp BruteForceSums
string className = "BruteForceSums";
writer.WriteLine("class {0}: DynInvocation.IComputer ",
    className);
writer.WriteLine("{");

// Tạo ra hàm ComputeSum
writer.WriteLine("\tpublic double ComputeSum()");
writer.WriteLine("\t{");
writer.WriteLine("\t// Brute force sum method");
writer.WriteLine("\t// For Value = {0}", theVal);

// viết ra kiểu cộng cụ thể
writer.Write("\treturn 0");
for (int i=1; i <= theVal; i++)
{
    writer.Write("+ {0}", i);
}
writer.WriteLine(";"); // kết thúc hàm
writer.WriteLine("\t"); // end method
writer.WriteLine("}"); // end class

writer.Close(); // đóng lại writer
s.Close();      // đóng lại stream

// Xây dựng tập tin
ProcessStartInfo psi = new ProcessStartInfo();
psi.FileName = "cmd.exe";
string compileString = "/c csc /optimize+ ";
compileString += "/r:\"DynInvocation.exe\"";
compileString += "/target:library ";
compileString += "{0}.cs > compile.out";

psi.Arguments = String.Format(compileString, fileName);
psi.WindowStyle = ProcessWindowStyle.Minimized;

Process proc = Process.Start(psi);
proc.WaitForExit(); // chờ ít nhất 2 giây

Assembly a = Assembly.LoadFrom(fileName + ".dll");
theComputer =
    (IComputer)a.CreateInstance(className);
File.Delete(fileName + ".cs"); // dọn dẹp
}

IComputer theComputer = null;

}

public class TestDriver
{

```

```

[STAThread]
static void Main()
{
    const int val = 200; // 1..200
    const int iterations = 100000;
    double result = 0;

    // Cho chạy benchmark
    MyMath m = new MyMath();
    DateTime startTime = DateTime.Now;

    for(int i=0; i<iterations; i++)
    {
        result = m.DoSumLooping(val);
    }
    TimeSpan elapsed = DateTime.Now - startTime;

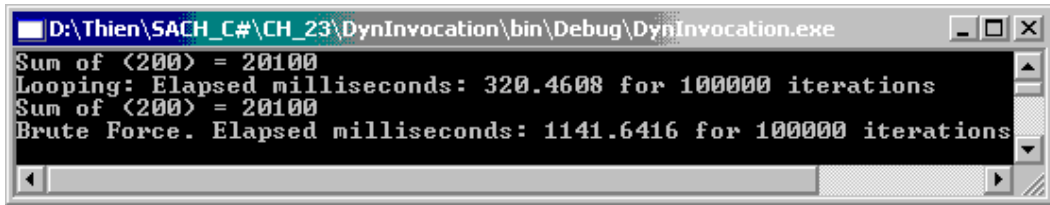
    Console.WriteLine("Sum of ({0}) = {1}", val, result);
    Console.WriteLine("Looping: Elapsed milliseconds: " +
        elapsed.TotalMilliseconds + " for {0} iterations",
        iterations);

    // Cho chạy kiểu Reflection
    ReflectionTest t = new ReflectionTest();
    startTime = DateTime.Now;
    for (int i=0; i<iterations; i++)
    {
        result = t.DoSum(val);
    }
    elapsed = DateTime.Now - startTime;
    Console.WriteLine("Sum of ({0}) = {1}", val, result);
    Console.WriteLine("Brute Force. Elapsed milliseconds: " +
        elapsed.TotalMillisecond + " for {0} iterations",
        iterations);

    Console.ReadLine();
}
}
}

```

Hình 4-18 cho thấy kết xuất lần này khả quan hơn một chút. Nhưng bạn có thể cải tiến tốc độ bằng cách sử dụng Reflection.Emit.



Hình .4-18: Kết xuất Thí dụ 4-15

#### 4.3.4.3 *Dynamic Invocation sử dụng Reflection Emit*

Tới đây, coi như bạn tạo được một assembly on the fly bằng cách viết đoạn mã nguồn lên đĩa, rồi sau đó cho biên dịch đoạn mã nguồn này. Tiếp theo, bạn cho triệu gọi động hàm hành sự bạn muốn dùng từ assembly này, đã được biên dịch trên đĩa. Như vậy sẽ tốn nhiều công sức (overhead), và bạn đã thực hiện được việc gì? Khi bạn viết tập tin lên đĩa, bạn có bộ mã nguồn mà bạn có thể biên dịch, và khi bạn biên dịch bạn có những IL op code trên đĩa mà bạn có thể yêu cầu .NET Framework cho chạy.

Reflection emit cho phép bạn nhảy bỏ một vài bước và chỉ “emit” (phát sóng” trực tiếp các op-code. Đây có nghĩa là viết mã assembly trực tiếp từ chương trình C#, rồi cho triệu gọi hàm kết quả.

Giống như với các thí dụ đi trước, bạn tạo một hằng (200) để lấy tổng, và số lần lặp lại (1.000.000). Và bạn tạo lớp **MyMath** như là benchmark.

Một lần nữa, bạn có lớp **ReflectionTest**, và một lần nữa bạn triệu gọi hàm **DoSum**, trao qua **theValue**.

```
ReflectionTest t = new ReflectionTest();
result = t.DoSum(val);
```

Bản thân **DoSum()** không thay đổi gì:

```
public double DoSum(int theValue)
{
    if (theComputer == null)
    {
        GenerateCode(theValue);
    }
    return (theComputer.ComputeSum());
}
```

Như bạn có thể thấy, lần nữa bạn sẽ dùng giao diện, nhưng lần này bạn không viết một tập tin lên đĩa.

Hàm **GenerateCode** hoàn toàn khác hẳn. Thay vì viết tập tin lên đĩa và cho biên dịch, bạn sẽ triệu gọi hàm phụ trợ **EmitAssembly** để nhận về một assembly. Sau đó bạn tạo một thể hiện của assembly này và cast thể hiện này về giao diện của bạn:

```
public void GenerateCode(int theValue)
{
    Assembly a = EmitAssembly(theValue);
    theComputer = (IComputer) a.CreateInstance("BruteForceSums");
}
```

Như bạn có thể mong đợi, điểm lý thú là trong hàm hành sự **EmitAssembly**:

```
private Assembly EmitAssembly(int theValue)
```

Trị mà bạn trao qua là tổng mà bạn muốn tính ra. Muốn thấy sức mạnh của ref emit, bạn sẽ tăng trị từ 200 lên 2000.

Điều đầu tiên phải làm trong **EmitAssembly** là tạo một đối tượng kiểu dữ liệu **AssemblyName** và trao cho đối tượng này tên “**DoSumAssembly**”:

```
AssemblyName asn = new AssemblyName();
asn.Name = "DoSumAssembly";
```

Đối tượng **AssemblyName** là một đối tượng mô tả trọn vẹn lý lịch duy nhất của một assembly. Như bạn đã biết một assembly được nhận diện bởi một tên thân thiện (**DoSumAssembly**), một số phiên bản, một cặp mục khoá private/public và một culture được hỗ trợ.

Với đối tượng này trong tay, bạn có thể tạo một đối tượng **AssemblyBuilder** mới. Muốn thế, bạn cho triệu gọi **hàmDefineDynamicAssembly** đối với domain hiện hành bằng cách gọi hàm hành sự static **GetDomain()** thuộc đối tượng **Thread**. Các thông số của **GetDomain()** là đối tượng **AssemblyName** bạn vừa mới tạo ra và một trị enum **AssemblyBuilderAccess** (một trong những trị **Run**, **RunAndSave**, hoặc **Save**). Bạn sẽ dùng **Run** trong trường hợp này cho biết assembly có thể được cho chạy khỏi cất trữ.

```
// Tạo một assembly mới
AssemblyBuilder newAsm = Thread.GetDomain().DefineDynamicAssembly(
    asn, AssemblyBuilderAccess.Run);
```

Với một đối tượng **AssemblyBuilder** mới được tạo ra, bạn sẵn sàng tạo ra đối tượng **ModuleBuilder**. Công việc của **ModuleBuilder**, không ngạc nhiên cho lắm, là tạo động một module. Module đã được đề cập đến ở chương 3, “Tìm hiểu về Assembly và cơ chế Version”. Bạn cho triệu gọi hàm **DefineDynamicModule**, trao qua tên của hàm hành sự bạn muốn tạo ra:

```
ModuleBuilder newMod = newAsm.DefineDynamicModule("Sum");
```



Bây giờ, dựa trên mod nào đó, bạn có thể định nghĩa một lớp public và nhận về một đối tượng **TypeBuilder**. TypeBuilder là lớp gốc để dùng điều khiển việc tạo động các lớp. Với một đối tượng TypeBuilder, bạn có thể định nghĩa các lớp và thêm các vùng mục tin và hàm hành sự:

```
// Định nghĩa một public class BruteForceSums trên assembly
TypeBuilder myType = new Mod.DefineType("BruteForceSums",
                                         TypeAttributes.Public);
```

Bây giờ, bạn đã sẵn sàng đánh dấu lớp mới như là thi công giao diện **IComputer**.

```
// Đánh dấu lớp như là thi công IComputer
myType.AddInterfaceImplementation(typeof(IComputer));
```

Trước khi bạn bắt đầu tạo hàm hành sự ComputeSum, trước tiên bạn phải setup một bản dãy các thông số. Vì bạn không có thông số nên bản dãy có bề dài bằng zero, rồi sau đó bạn tạo một đối tượng **Type** dùng để return type đối với hàm hành sự:

```
// Định nghĩa một hàm
Type[] paramTypes = new Type[0];
Type returnType = typeof(int);
```

Bạn đã sẵn sàng tạo hàm hành sự **ComputeSum**. Hàm hành sự **TypeBuilder.DefineMethod** sẽ tạo hàm hành sự và trả về một đối tượng kiểu dữ liệu MethodBuilder mà bạn sẽ dùng để kết sinh IL code:

```
MethodBuilder simpleMethod = myType.DefineMethod("ComputeSum",
                                                  MethodAttributes.Public | MethodAttributes.Virtual,
                                                  returnType,
                                                  paramTypes);
```

Bạn trao qua tên hàm hành sự (**ComputeSum**), flags bạn muốn đặt để (**public** và **virtual**), kiểu dữ liệu trả về (**int**) và **paramTypes** (bản dãy bề dài zero).

Sau đó bạn sử dụng đối tượng MethodBuilder mà bạn đã tạo ra để nhận lấy một đối tượng **ILGenerator**:

```
// Đi lấy một IL Generator
ILGenerator gen = simpleMethod.GetILGenerator();
```

Có trong tay một đối tượng ILGenerator, bạn đã sẵn sàng emit op-code. Đây chính là những op-code mà C# compiler sẽ tạo ra.

Trước tiên, emit trị 0 đối với stack. Sau đó rảo qua các trị số bạn muốn thêm vào (1 đến 2000), thêm mỗi lần vào stack, kết quả ghi lên stack:

```
// Emit IL
gen.Emit(OpCodes.Ldc_I4, 0);
for (int i=0; i<= theValue; i++)
{
    gen.Emit(OpCodes.Ldc_I4, i);
    gen.Emit(OpCodes.Add);
}
```

Trị còn lại trên stack là tổng bạn muốn, do đó trả nó về:

```
gen.Emit(OpCodes.Ret);
```

Tiếp theo bạn tạo một đối tượng **MethodInfo** lo mô tả hàm hành sự:

```
MethodInfo computeSumInfo=typeof(IComputer).GetMethod("ComputeSum");
```

Bây giờ bạn phải khai báo phần thi công hàm hành sự này. Bạn triệu gọi hàm **DefineMethodOverride** đối với đối tượng **TypeBuilder** mà bạn đã tạo ra trước đó, trao qua đối tượng **MethodBuilder** bạn đã tạo ra kèm theo đối tượng **MethodInfo** bạn cũng vừa tạo ra:

```
myType.DefineMethodOverride(simpleMethod, computeSumInfo);
```

Thế là xong; tạo xong lớp và trở về assembly:

```
myType.CreateType();
return newAsm;
```

Thí dụ 4-16 là toàn bộ chương trình sử dụng **Reflection.Emit**

#### ***Thí dụ 4-16: Dynamic invocation sử dụng Reflection.Emit***

```
using System;
using System.Diagnostics;
using System.Reflection;
using System.IO;
using System.Reflection.Emit;
using System.Threading;
namespace ReflectionEmit
{
    // Lớp này dùng làm thước đo đối với kiểu vòng lặp
    public class MyMath
    {
        // tính tổng dùng vòng lặp
        public int DoSumLooping(int initialVal)
        {
            int result = 0;
            for(int i=0; i<=initialVal;i++)
```

```

        {
            result += i;
        }
        return result;
    }
}

// Khai báo giao diện
public interface IComputer
{
    int ComputeSum();
}

public class ReflectionTest
{
    // Hàm private này emit assembly sử dụng op codes
    private Assembly EmitAssembly(int theValue)
    {
        // Tạo một tên assembly
        AssemblyName asn = new AssemblyName();
        asn.Name = "DoSumAssembly";

        // Tạo một assembly mới với một module
        AssemblyBuilder newAsm =
            Thread.GetDomain().DefineDynamicAssembly(
                asn, AssemblyBuilderAccess.Run);
        ModuleBuilder newMod = newAsm.DefineDynamicModule("Sum");

        // Định nghĩa một public class BruteForceSums trên assembly
        TypeBuilder myType = newMod.DefineType("BruteForceSums",
            TypeAttributes.Public);

        // Đánh dấu lớp như là thi công IComputer
        myType.AddInterfaceImplementation(typeof(IComputer));

        // Định nghĩa một hàm
        Type[] paramTypes = new Type[0];
        Type returnType = typeof(int);
        MethodBuilder simpleMethod = myType.DefineMethod(
            "ComputeSum",
            MethodAttributes.Public | MethodAttributes.Virtual,
            returnType,
            paramTypes);

        // Đi lấy một IL Generator dùng emit IL bạn muốn
        ILGenerator gen = simpleMethod.GetILGenerator();

        // Emit IL. Cho ẩn zero vào stack
        gen.Emit(OpCodes.Ldc_I4, 0);

        // Đối với mỗi i nhỏ thua theValue, ẩn i như là hằng
        // rồi cộng hai số nằm đầu stack.
        // Tổng cộng để lại trên stack
        for (int i=0; i<= theValue; i++)
    }
}

```

```

        {
            gen.Emit(OpCodes.Ldc_I4, i);
            gen.Emit(OpCodes.Add);
        }

        // trả về trị
        gen.Emit(OpCodes.Ret);

        // Gói ghém thông tin liên quan đến hàm hành sự
        // rồi cung cấp truy xuất vào metadata CỦA hàm hành sự
        MethodInfo computeSumInfo =
            typeof(IComputer).GetMethod("ComputeSum");

        // cho biết thi công hàm hành sự
        myType.DefineMethodOverride(simpleMethod, computeSumInfo);

        // Tạo kiểu dữ liệu
        myType.CreateType();
        return new Asm;
    }

    // kiểm tra xem giao diện có null hay không
    // nếu null, thì gọi Setup
    public double DoSum(int theValue)
    {
        // nếu không có qui chiếu về lớp thì phải tạo ra
        if (theComputer == null)
        {
            GenerateCode(theValue);
        }
        // triệu gọi hàm thông qua giao diện
        return (theComputer.ComputeSum());
    }

    // emit assembly, tạo một thẻ hiện
    // rồi đi lấy giao diện
    public void GenerateCode(int theValue)
    {
        Assembly a = EmitAssembly(theValue);
        theComputer = (IComputer)
            a.CreateInstance("BruteForceSums");
    }

    // biến thành viên private
    IComputer theComputer = null;
}
public class TestDriver
{
    [STAThread]

```

```

public static void Main()
{
    const int val = 2000; // 1..2000
    const int iterations = 1000000;
    double result = 0;

    // Cho chạy benchmark
    MyMath m = new MyMath();
    DateTime startTime = DateTime.Now;

    for(int i=0; i<iterations; i++)
    {
        result = m.DoSumLooping(val);
    }
    TimeSpan elapsed = DateTime.Now - startTime;

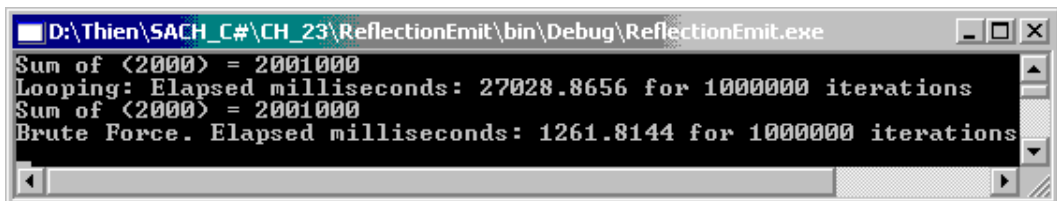
    Console.WriteLine("Sum of ({0}) = {1}", val, result);
    Console.WriteLine("Looping: Elapsed milliseconds: " +
        elapsed.TotalMilliseconds + " for {0} iterations",
        iterations);

    // Cho chạy kiểu Reflection
    ReflectionTest t = new ReflectionTest();
    startTime = DateTime.Now;
    for (int i=0; i<iterations; i++)
    {
        result = t.DoSum(val);
    }
    elapsed = DateTime.Now - startTime;
    Console.WriteLine("Sum of ({0}) = {1}", val, result);
    Console.WriteLine("Brute Force. Elapsed milliseconds: " +
        elapsed.TotalMilliseconds + " for {0} iterations",
        iterations);

    Console.ReadLine();
}
}

```

Hình 4-19 cho thấy kết quả. Lần này chạy theo kiểu cụ trau nhanh hơn là với vòng lặp.



Hình 4-19: Kết xuất Thí dụ 4-16 sử dụng Reflect.Emit

## Chương 5

# Marshaling và Remoting

Những ngày mà các chương trình được tích hợp chạy trên cùng một process duy nhất và trên một máy duy nhất đã qua rồi, quá lỗi thời. Ngày nay, các chương trình thường bao gồm những thành phần (được gọi là “cấu kiện”, component) phức tạp chạy trên nhiều process (giống như những “đường đua”) xuyên qua hệ thống mạng. Web làm cho việc chạy các ứng dụng mạng tính phát tán (distributed application) ngày càng dễ dàng hơn theo vô số cách khó tưởng tượng nổi chỉ một vài năm về trước, và chiều hướng là ngày càng đi đến việc phân tán trách nhiệm.

Chiều hướng thứ hai là tập trung khía cạnh business logic lên các server đồ sộ. Ngoài mặt xem ra nghịch lý, nhưng trong thực tế các đối tượng mạng tính business (sản xuất kinh doanh) thì theo hướng tập trung, trong khi giao diện người sử dụng thì lại phân tán.

Tác dụng rõ ràng nhất là các đối tượng có khả năng nói chuyện với nhau mà ở cách xa nhau. Các đối tượng chạy trên server lo thụ lý giao diện web của người sử dụng cần có khả năng tương tác với các đối tượng business nằm ở những server được tập trung tại trụ sở chính của công ty.

Tiến trình di chuyển một đối tượng xuyên qua một ranh giới được gọi là *remoting* (hành trình đi xa). Ranh giới hiện hữu ở nhiều cấp độ trừu tượng khác nhau trong chương trình của bạn. Ranh giới rõ ràng nhất là giữa các đối tượng chạy trên những máy tính khác nhau.

Tiến trình chuẩn bị một đối tượng được gửi đi xa được gọi là *marshaling*<sup>13</sup>. Trên một máy đơn độc các đối tượng có thể được marshal xuyên phạm trù, xuyên app domain hoặc xuyên ranh giới process.

Một *process* thực chất là một ứng dụng đang chạy. Nếu một đối tượng trên trình soạn thảo văn bản muốn tương tác với một đối tượng trên bảng tính, đối tượng này phải liên lạc xuyên ranh giới.

Các process được chia thành *application domain* (thường được gọi là “app domain”), và app domain lại được chia thành *context* (phạm trù). App domain hành động như là

---

<sup>13</sup> Marshaling là “cho vào nề nếp quân ngũ”. Bạn có thể hình dung marshaling như là việc các nhà sản xuất đóng hàng vào container chuyên đi theo đường hàng hải, hoặc đóng gói gửi theo đường bưu điện.

những process nhẹ cân, còn context sẽ tạo ra những ranh giới theo đây các đối tượng cùng chia sẻ các qui tắc tương tự có thể nằm trong lòng ranh giới. Có lúc, các đối tượng sẽ được marshal xuyên qua cả ranh giới context lẫn ranh giới app domain, cũng như xuyên qua ranh giới process và máy tính.

Khi một đối tượng nằm ở xa, nó có vẻ được gọi đi thông qua đường dây nối liền máy tính này qua máy tính kia. Nếu bạn ở phía kia đường dây bạn có thể nghĩ rằng bạn nhìn thấy đối tượng và đang nói chuyện với đối tượng. Nhưng thật ra bạn không nói chuyện với đối tượng mà là đang nói chuyện với một *proxy*, hoặc một cái mô phỏng. Nhiệm vụ của proxy là tiếp nhận thông điệp của bạn rồi chuyển cho đối tác. Ngoài ra, giữa bạn và đối tác còn có vô số “sink”.

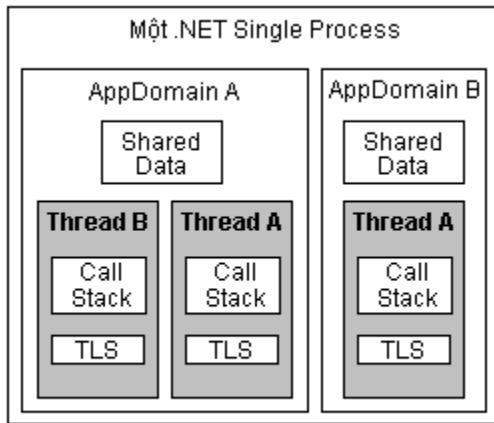
Sink là một đối tượng có nhiệm vụ tăng cường cơ chế liên lạc. Khi phía đối tác chuyển một cái gì không hợp lệ theo cơ chế, thì sink sẽ cho ngưng ngay việc liên lạc. Khi phía đối tác trả lời, thì nó chuyển câu trả lời cho những sink khác nhau cho đến khi tới tay proxy, và proxy sẽ nói chuyện với bạn.

Việc liên lạc chuyển thông điệp của bạn sẽ được thực hiện thông qua một *channel* (kênh liên lạc). Công việc của kênh là biết thể nào chuyển thông điệp từ nơi này qua nơi kia. Channel làm việc với một bộ định dạng (*formatter*). Formatter bảo đảm là thông điệp theo đúng dạng thức. Formatter giữ vai trò lặng lẽ của một thông dịch viên nói ngôn ngữ của bạn và ngôn ngữ của phía đối tác, cả hai có thể là không cùng một ngôn ngữ.

Chương này sẽ minh họa làm thế nào các đối tượng có thể được marshal xuyên qua các ranh giới khác nhau, và làm thế nào proxy và stub (là đối tác của proxy) có thể tạo một ảo tưởng là đối tượng của bạn chạy xuyên qua đường dây mạng đến tận máy tính của bạn hoặc đi vòng quanh thế giới. Ngoài ra, chương này sẽ giải thích vai trò của formatter, channel và sink, và cách dùng các khái niệm này trong lập trình.

## 5.1 Application Domains

Một process thực chất là một ứng dụng đang chạy. Mỗi ứng dụng .NET đều chạy trong process riêng của mình (giống như đường đua thể thao). Nếu bạn cho mở cùng lúc Word, Excel và Visual Studio .NET, thì bạn có 3 process đang chạy. Mỗi process lại phân nhỏ thành một hoặc nhiều application domain, mỗi application domain (gọi tắt là AppDomain) sẽ hoàn toàn được cách ly khỏi các AppDomain khác trong lòng process. Một app domain hành động như là một process nhưng sử dụng ít nguồn lực hơn. Các ứng dụng chạy trên AppDomain khác nhau sẽ không có khả năng chia sẻ bất cứ thông tin nào (biên toàn cục hoặc vùng mục tin static) trừ phi chúng tuân thủ .NET remoting protocol. Hình 5-01 cho thấy toàn cảnh Process, AppDomain và Thread.



**Hình 5-1: Cấu trúc Process, AppDomain**

App domain có thể được khởi động hoặc cho ngưng một cách hoàn toàn độc lập. Chúng được xem như là an toàn, “nhẹ cân” (nghĩa là không sử dụng nhiều nguồn lực) và linh hoạt cũng như mang tính “chấp nhận sai lầm” (fault tolerance). Nếu bạn khởi động một đối tượng trong một app domain thứ hai và nó “sụp bả chề” (crash), thì nó sẽ cho đi đong app domain nhưng toàn bộ chương trình thì không hề hấn gì. Bạn có thể tưởng tượng là Web Server phải sử dụng app domain để cho chạy đoạn mã của người sử dụng, và nếu đoạn mã có vấn đề, thì Web Server có thể duy trì công tác.

Một app domain sẽ được gói ghém bởi một thể hiện của lớp **AppDomain**. Lớp này gồm một số hàm hành sự và thuộc tính. Bảng 5-01 liệt kê một số quan trọng:

**Bảng 5-01: Các thành viên cốt lõi của lớp *System.AppDomain***

Các thành viên	Mô tả
<b>CurrentDomain</b>	Thuộc tính public static này trả về app domain hiện hành đối với mạch trình hiện hành.
<b>CreateDomain()</b>	Hàm overloaded public static này tạo một app domain mới trong process hiện hành.
<b>GetCurrentThreadID()</b>	Hàm public static trả về mã nhận diện mạch trình hiện hành.
<b>Unload()</b>	Hàm public static lo gỡ bỏ AppDomain được khai báo.
<b>FriendlyName</b>	Thuộc tính public trả về tên thân thiện đối với AppDomain này.
<b>DefineDynamicAssembly()</b>	Hàm overloaded public cho phép định nghĩa một dynamic assembly trong AppDomain hiện hành.
<b>ExecuteAssembly()</b>	Hàm public lo thi hành assembly được chỉ định.
<b>GetData()</b>	Hàm public cho phép đi lấy trị được trữ trong app domain hiện hành.
<b>Load()</b>	Hàm public lo nạp một assembly vào app domain hiện hành.
<b>SetAppDomainPolicy()</b>	Hàm public lo đặt để cơ chế an toàn (security policy) đối với AppDomain hiện hành.
<b>SetData()</b>	Hàm public cho phép đưa dữ liệu vào thuộc tính app domain được chỉ định.



Ngoài ra, AppDomain cũng hỗ trợ vô số tình huống khác nhau, bao gồm **AssemblyLoad**, **AssemblyResolve**, **ProcessExit**, và **ResourceResolve**, được phát pháo khi assembly được tìm thấy, nạp, chạy và gỡ bỏ.

Mỗi process đều có một app domain khởi sự, và có thể có những app domain bổ sung khi bạn tạo ra chúng. Mỗi app domain hiện hữu đúng trong một process. Mãi tới nay, tất cả các chương trình trong tập sách này chỉ có một app domain đơn độc; app domain mặc nhiên. Mỗi process đều có app domain mặc nhiên riêng. Trong phần lớn các chương trình bạn viết ra, bạn chỉ cần app domain mặc nhiên là đủ.

Tuy nhiên, đôi lúc một domain đơn độc là không đủ. Có thể bạn muốn tạo một app domain thứ hai (sử dụng hàm **CreateDomain()** nếu bạn muốn cho chạy một thư viện viết bởi một lập trình viên khác). Có thể, bạn không tin tưởng cho lắm thư viện này và bạn muốn cách ly nó khỏi domain riêng của bạn để khi một hàm hành sự nào đó trên thư viện sụm thì chỉ domain bị cách ly mới bị ảnh hưởng mà thôi.

Cũng có thể là thư viện kia đòi hỏi một môi trường an ninh khác đi; tạo một app domain thứ hai cho phép hai môi trường an ninh chung sống hoà bình. Mỗi app domain có riêng cho mình một môi trường an ninh, và như vậy app domain được dùng như là ranh giới an ninh.

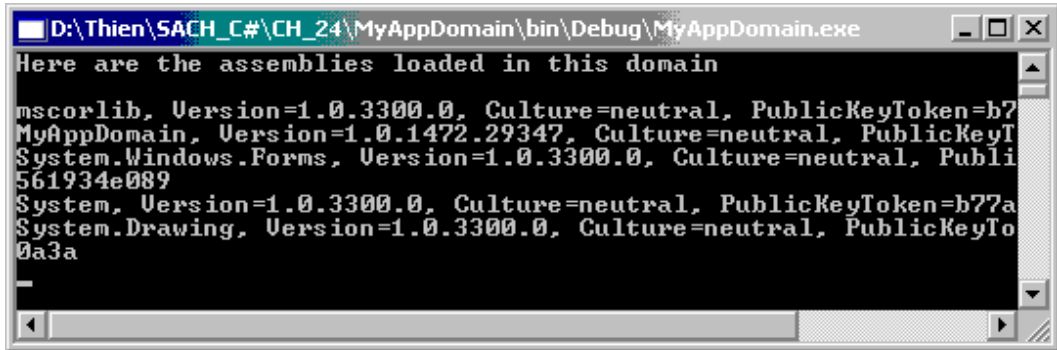
App domain không phải là mạch trình <sup>14</sup>(thread) và phải phân biệt khỏi mạch trình. Một mạch trình chỉ hiện hữu trong một app domain vào một lúc nào đó, mà một mạch trình có thể truy cập (và báo cáo) vào app domain nó đang thi hành. App domain được dùng để cách ly các ứng dụng; trong lòng một app domain có thể có nhiều mạch trình cùng hoạt động vào một lúc nào đó. Xem lại hình 5-01.

## 5.1.1 Chơi một chút với AppDomain

Để xem app domain hoạt động thế nào, ta thử xem thí dụ 5-01 sau đây, với kết xuất là hình 5-02:

---

<sup>14</sup> Có người dịch là “tiểu trình”.



**Hình 5-02: Khảo sát các assembly được nạp vào**

### ***Thí dụ 5-01: Khảo sát các assembly được nạp vào một AppDomain***

```
using System;
using System.Windows.Forms;
using System.Reflection; // cần namespace này để làm việc với
                        // kiểu dữ liệu Assembly
namespace MyAppDomain
{
    public class MyAppDomain
    {
        public static void PrintAllAssemblies()
        {
            // Yêu cầu AppDomain liệt kê tất cả các assembly
            // được nạp vào
            AppDomain ad = AppDomain.CurrentDomain;
            Assembly[] loadedAss = ad.GetAssemblies();
            Console.WriteLine("Here are the assemblies loaded in this
                              domain\n");

            // In ra tên trọn vẹn của mỗi assembly
            foreach(Assembly a in loadedAss)
            {
                Console.WriteLine(a.FullName);
            }
        }

        public static int Main(string[] args)
        {
            // Ép nạp assembly Windows Forms
            MessageBox.Show("Loaded System.Windows.Forms.dll");
            PrintAllAssemblies();
            Console.ReadLine(); // dùng ngăn cửa sổ console biến đi
            return 0;
        }
    }
}
```

Bạn để ý, trong chương trình này bạn sử dụng đến namespace **System.Reflection**, lo định nghĩa kiểu dữ liệu **Assembly** mà hàm **PrintAllAssemblies()** sẽ dùng đến. **Assembly** đã được đề cập đến ở chương 4, “Tìm hiểu về Attribute và Reflection”, đi trước.

Hàm static **PrintAllAssemblies** nhận lấy qui chiếu về AppDomain hiện hành và liệt kê ra những assembly hiện đang “đóng quân” trên app domain. Bạn để ý hàm hành sự **Main()** cho hiển thị một message box yêu cầu assembly resolver nạp assembly System.Windows.Forms.dll (đến phiên DLL này nạp các assembly được qui chiếu khác). Hình 5-02 cho thấy kết xuất.

## 5.1.2 Một thí dụ sử dụng AppDomain

Muốn xem app domain hoạt động thế nào, ta thử dàn dựng một thí dụ. Giả sử ta muốn chương trình hiển lộ một lớp **Shape**, nhưng ở trên một app domain thứ hai.

**Bạn để ý:** Không có lý do gì chính đáng phải đưa lớp **Shape** vào một app domain thứ hai, ngoại trừ việc muốn minh hoạ các kỹ thuật này hoạt động thế nào. Tuy nhiên, đối với những đối tượng phức tạp hơn, có thể ta cần một app domain thứ hai cung cấp một môi trường an ninh khác. Ngoài ra, khi tạo ra những lớp mới, có thể bạn sẽ đưa vào những hành xử nhiều rủi ro, và để cho an toàn có thể bạn bắt đầu các lớp mới này trên một app domain thứ hai.

Thông thường, bạn nạp lớp **Shape** từ một assembly riêng biệt, nhưng để cho đơn giản thí dụ bạn chỉ cần đưa định nghĩa của lớp **Shape** vào cùng mã nguồn như với các đoạn mã khác. Ngoài ra, trên một môi trường sản xuất, có thể bạn cho chạy các hàm hành sự của lớp **Shape** trên một mạch trình riêng biệt. Nhưng để cho đơn giản, tạm thời bạn quên đi mạch trình. Threading sẽ được đề cập đến ở chương 6, “Mạch trình và Đồng bộ hoá”, đi sau. Như vậy, thí dụ sẽ đơn giản hơn và ta chỉ tập trung vào những chi tiết tạo và sử dụng app domain cũng như marshal các đối tượng xuyên ranh giới phân chia các app domain.

### 5.1.2.1 Tạo và Sử dụng App Domain

Muốn tạo một app domain mới bạn cho triệu gọi hàm static **CreateDomain()** thuộc lớp **AppDomain**:

```
AppDomain ad2 = AppDomain.CreateDomain("Shape Domain", null, null);
```

Lệnh trên tạo mới một app domain ad2 mang tên thân thiện Shape Domain. Tên thân thiện chỉ dành cho lập trình không cần biết đến cách biểu diễn nội tại của app domain.

Bạn có thể kiểm tra tên thân thiện của app domain bạn đang làm việc dựa trên thuộc tính **System.AppDomain.CurrentDomain.FriendlyName**.

Một khi bạn đã hiển lộ một đối tượng **AppDomain**, bạn có thể tạo những thể hiện các lớp, giao diện, v.v.. sử dụng hàm hành sự **CreateInstance()**. Sau đây là dấu ấn của hàm hành sự:

```
public ObjectHandle CreateInstance(  
    string assemblyName,  
    string typeName,  
    bool ignoreCase,  
    BindingFlags bindingAttr,  
    Binder binder,  
    object[] args,  
    CultureInfo culture,  
    object[] activationAttributes,  
    Evidence securityAttributes);
```

Và đây là cách dùng hàm hành sự này:

```
ObjectHandle oh = ad2.CreateInstance(  
    "ProgCSharp",           // tên assembly  
    "ProgCSharp.Shape",    // tên kiểu dữ liệu với namespace  
    false,                 // ignore case  
    System.Reflection.BindingFlags.CreateInstance, // flag  
    null,                  // binder  
    new object[] {3, 5},   // args  
    null,                 // culture  
    null,                 // activation attribute  
    null );               // security attribute
```

Thông số đầu tiên (**ProgCSharp**) là tên của assembly, còn thông số thứ hai (**ProgCSharp.Shape**) là tên lớp. Tên phải được “fully qualified” kèm theo namespace.

Một đối tượng *binder* cho phép việc kết nối động (dynamic binding) của một assembly vào lúc chạy. Công việc của binder là cho phép bạn chuyển giao thông tin liên quan đến đối tượng bạn muốn tạo, tạo đối tượng này cho bạn và gắn kết qui chiếu của bạn vào đối tượng này. Trong phần lớn trường hợp, kể cả thí dụ này, bạn sẽ sử dụng binder mặc nhiên bằng cách trao qua null.

Lẽ dĩ nhiên là bạn có thể viết một binder riêng cho mình, chẳng hạn bằng cách kiểm tra mã nhận diện ID so với quyền hạn truy cập đối với một căn cứ dữ liệu rồi chuyển hướng việc gắn kết qua một đối tượng khác, dựa trên mã nhận diện hoặc quyền hạn truy cập của bạn.

**Bạn để ý:** Từ “binding”, gắn kết, thường ám chỉ việc gắn liền một tên đối tượng vào một đối tượng. “Dynamic binding” ám chỉ khả năng việc gắn kết được thực hiện khi chúng ta đang chạy, so với khi đang thiết kế. Trong thí dụ này, đối tượng **Shape** được gắn kết vào biến thể hiện (instance variable) vào lúc chạy, thông qua hàm **CreateInstance()**.

Binding flags giúp binder hành xử một cách tinh tế hơn vào lúc gắn kết. Trong thí dụ này, bạn sử dụng trị **CreateInstance** của enum **BindingFlags**. Thông thường binder mặc nhiên đi tìm những lớp public để gắn kết, nhưng bạn có thể thêm flag cho phép đi tìm những lớp private nếu quyền hạn truy cập cho phép.

Khi bạn gắn kết một assembly vào lúc chạy, bạn không khai báo assembly phải nạp vào lúc biên dịch mà xác định assembly nào bạn muốn theo lập trình và gắn kết biến của bạn vào assembly này khi chương trình chạy.

Hàm constructor mà chúng tôi triệu gọi nhận hai số nguyên, phải được đưa vào một bản dãy đối tượng (**new object[] {3,5}**). Bạn có thể trao null đối với thông tin culture vì chúng tôi chọn trị culture mặc nhiên (**en**) và cũng sẽ không khác thông tin liên quan đến activation attribute và security attribute

Đối tượng mà bạn nhận về sẽ là một *object handle* (mục quản đối tượng). Đây là một **type** được dùng để trao một đối tượng (trong tình trạng được bao bọc - wrapped state) giữa nhiều app domain không nạp metadata đối với các đối tượng được bao bọc trong mỗi đối tượng theo đây **ObjectHandle** di chuyển. Bạn có thể đi lấy bản thân đối tượng hiện thời bằng cách triệu gọi hàm **UnWrap()** đối với mục quản đối tượng, và cho ép kiểu đối tượng kết xuất lên kiểu dữ liệu hiện thời - trong trường hợp này là **Shape**.

Hàm **CreateInstance()** cho phép tạo đối tượng trong một app domain mới. Nếu bạn muốn tạo mới đối tượng thông qua **new**, nó sẽ được tạo trong app domain hiện hành.

### 5.1.2.2 Marshalling xuyên ranh giới App Domain

Bạn đã tạo một đối tượng **Shape** trong Shape domain, nhưng bạn truy cập nó thông qua một đối tượng **Shape** trong domain nguyên thủy. Muốn truy xuất đối tượng shape trong một domain khác, bạn phải marshal đối tượng xuyên qua ranh giới của domain.

Marshalling là tiến trình chuẩn biến một đối tượng trước khi chuyển nó đi xuyên qua một ranh giới. Giống như khi bạn gói một món đồ cho ai đó ở tận đâu đó qua bưu điện, bạn phải cho đóng gói theo đúng tiêu chuẩn và ghi đầy đủ chi tiết trên gói hàng. Hành động này được gọi là marshalling. Marshalling có thể được thực hiện theo trị (by value) hoặc theo qui chiếu (by reference). Khi một đối tượng được marshal theo trị, thì một bản sao sẽ được thực hiện và được chuyển đi. Với bản sao này trên tay, bạn có thể làm gì tùy thích (nhặt tu, thêm bớt v.v..) nhưng bản gốc nguyên thủy sẽ không hề hấn gì.

Marshalling theo qui chiếu, nghĩa là bản gốc nằm tại chỗ, bạn không gởi đi bản sao mà lại một proxy. Khi bạn làm gì trên proxy, thì tác dụng sẽ ảnh hưởng lên bản gốc giống như là bạn thấy ngay bản gốc trước mặt mình.

### ***Tìm hiểu marshalling với proxy***

Khi bạn marshal theo qui chiếu, thì CLR sẽ cung cấp cho đối tượng phía triệu gọi một *transparent proxy* (TP). Công việc của TP là nhận về mọi thứ liên quan đến triệu gọi hàm của bạn (trị trả về, các thông số v.v..) khỏi stack và nhét nó vào một đối tượng có thi công giao diện **IMessage**. **IMessage** này sẽ được trao qua cho một đối tượng **RealProxy**.

**RealProxy** là một lớp abstract mà từ đây các proxy sẽ được dẫn xuất. Bạn có thể thi công proxy thật sự riêng của mình, hoặc bất cứ đối tượng khác trong process ngoại trừ TP. Proxy mặc nhiên thực thụ sẽ thụ lý **IMessage** đối với một loạt những đối tượng sink.

Bất cứ số lượng sink nào có thể được sử dụng tùy thuộc vào số cơ chế mà bạn muốn tăng cường, nhưng đúng sink chót trong chuỗi sink sẽ đưa **IMessage** vào một **Channel**. Kênh sẽ được chia thành kênh phía khách và kênh phía server và công việc của kênh là “đưa đồ” xuyên ranh giới. Kênh chịu trách nhiệm hiểu thấu nghi thức giao thông vận chuyển. Dạng thức hiện thời đối với một thông điệp khi xuyên biên giới sẽ được quản lý bởi một *formatter* (bộ định dạng). .NET Framework cung cấp hai loại formatter: loại thứ nhất mang tên **SOAP** (Simple Object Access Protocol), được xem là mặc nhiên đối với kênh HTTP, và loại thứ hai mang tên **Binary Formatter** được xem là mặc nhiên đối với kênh TCP/IP. Không ai cấm bạn tạo ra cho mình một formatter đối với kênh riêng của bạn nếu bạn khoái làm việc này.

Một khi thông điệp đã băng qua ranh giới, nó sẽ được tiếp đón bởi kênh phía server, và một formatter, lo tạo lại **IMessage** và chuyển cho một hoặc nhiều sink phía server. Sink cuối cùng trên chuỗi sink này là **StackBuilder**, lo nhận **IMessage**, trả về lại cho stack frame theo đây nó xuất hiện như là một triệu gọi hàm đối với server.

### ***Khai báo phương pháp marshalling***

Trong thí dụ kế tiếp, muốn minh họa sự phân biệt giữa marshal by value và marshal by reference, bạn cho biết đối tượng **Shape** sẽ được marshal theo qui chiếu, nhưng lại trao cho nó một biến thành viên kiểu dữ liệu **Point** mà bạn sẽ khai báo như là được marshal theo trị.

Bạn để ý mỗi lần bạn muốn tạo một đối tượng, mà đối tượng này lại có ý vượt biên thì bạn phải chọn cách nó sẽ được marshal thế nào. Thông thường các đối tượng không thể được marshal chi cả; bạn phải thân chinh cho biết đối tượng có biến marshal hay không, hoặc bằng trị hoặc bằng qui chiếu.

Cách dễ nhất khai báo cho biết một đối tượng được marshal theo trị là sử dụng attribute **[Serializable]**:

```
{Serializable}
public class Point
```

Khi một đối tượng được serialize, thì trạng thái nội tại của đối tượng sẽ được biến thành một stream dùng để marshal hoặc dùng cất trữ. Chi tiết về serialization đã được đề cập đến ở chương 1, “Xuất nhập dữ liệu & Sản sinh hằng loạt đối tượng”.

Cách dễ nhất khai báo cho biết một đối tượng được marshal theo qui chiếu là cho dẫn xuất lớp của nó từ **MarshalByRefObject**:

```
public class Shape: MarshalByRefObject
```

Lớp **Shape** sẽ chỉ có một biến thành viên, **upperLeft**. Biến này mang kiểu dữ liệu **Point**, lo cầm giữ tọa độ của góc bên trái phía trên của đối tượng shape. Hàm constructor của **Shape** sẽ được khởi gán bởi thành viên **Point**:

```
public Shape(int upperLeftX, int upperLeftY)
{
    Console.WriteLine("[{0}] {1}",
        System.AppDomain.CurrentDomain.FriendlyName,
        "Shape constructor");
    upperLeft = new Point(upperLeftX, upperLeftY);
}
```

Lớp **Shape** có một hàm hành sự, **ShowUpperLeft**, sự lo hiển thị vị trí của **upperLeft**:

```
public void ShowUpperLeft()
{
    Console.WriteLine("[{0}] Upper left: {1}, {2}",
        System.AppDomain.CurrentDomain.FriendlyName,
        upperLeft.X, upperLeft.Y);
}
```

Ngoài ra, lớp **Shape** còn cung cấp một hàm hành sự trả về biến thành viên **upperLeft**.

```
public Point GetUpperLeft()
{
    return upperLeft;
}
```

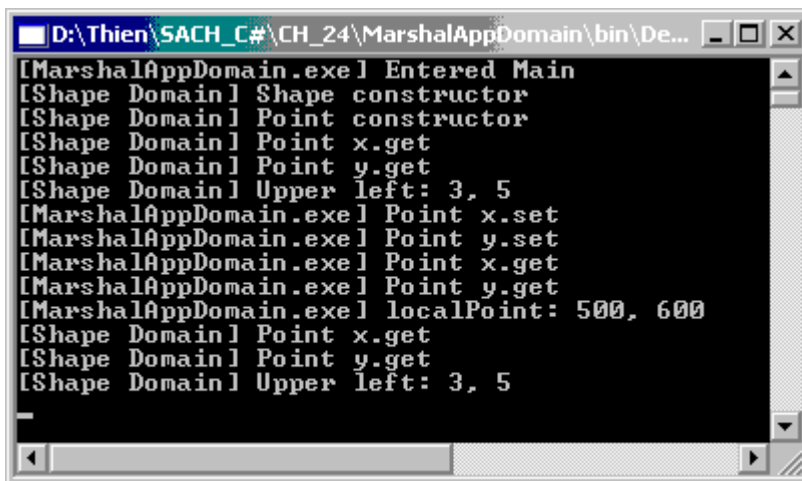
Lớp **Point** rất đơn giản. Nó có một hàm constructor lo khởi gán hai biến thành viên tọa độ và các hàm accessor đi lấy trị của chúng. Một khi bạn đã tạo ra đối tượng **Shape**, bạn có thể yêu cầu xem tọa độ:

```
s1.ShowUpperLeft(); // yêu cầu hiển thị tọa độ
```

Sau đó yêu cầu đối tượng trả về tọa độ upperLeft như là đối tượng **Point** mà bạn sẽ thay đổi tọa độ:

```
Point localPoint = s1.GetUpperLeft();
localPoint.X = 500;
localPoint.Y = 600;
```

Yêu cầu **Point** in ra tọa độ, và sau đó yêu cầu **Shape** in ra tọa độ của nó. Như vậy, việc thay đổi trên đối tượng **Point** có được phản ánh lên **Shape** hay không? Điều này tùy thuộc vào việc **Point** được marshal thế nào. Nếu đối tượng **Point** được marshal theo trị, thì đối tượng **localPoint** sẽ là một bản sao, và đối tượng **Shape** sẽ không bị ảnh hưởng bởi việc thay đổi trị của các biến của **localPoint**. Nếu ngược lại, bạn thay đổi đối tượng được marshal theo reference, thì bạn sẽ có một proxy đối với biến **upperLeft** hiện thời, và việc thay đổi này sẽ ảnh hưởng lên **Shape**. Thí dụ 5-02 minh họa các giải thích trên, và hình 5-03 là kết xuất:



Hình 5-03: Kết xuất thí dụ 5-02.

### Thí dụ 5-02: Marshalling xuyên biên giới app domain

```
using System;
using System.Runtime.Remoting;
using System.Reflection;

namespace MarshalAppDomain
```



```

{
    // đối với marshal by reference bạn cho comment out attribute
    // [Serializable] và uncomment base class
    [Serializable]
    public class Point //: MarshalByRefObject
    {
        public Point(int x, int y)
        {
            Console.WriteLine("[{0}] {1}",
                System.AppDomain.CurrentDomain.FriendlyName,
                "Point constructor");
            this.x = x;
            this.y = y;
        }
        public int X
        {
            get
            {
                Console.WriteLine("[{0}] {1}",
                    System.AppDomain.CurrentDomain.FriendlyName,
                    "Point x.get");
                return this.x;
            }
            set
            {
                Console.WriteLine("[{0}] {1}",
                    System.AppDomain.CurrentDomain.FriendlyName,
                    "Point x.set");
                this.x = value;
            }
        }

        public int Y
        {
            get
            {
                Console.WriteLine("[{0}] {1}",
                    System.AppDomain.CurrentDomain.FriendlyName,
                    "Point y.get");
                return this.y;
            }
            set
            {
                Console.WriteLine("[{0}] {1}",
                    System.AppDomain.CurrentDomain.FriendlyName,
                    "Point y.set");
                this.y = value;
            }
        }
        private int x;
        private int y;
    }

    // Lớp Shape được marshal theo qui chiếu
    public class Shape: MarshalByRefObject
    {
        public Shape(int upperLeftX, int upperLeftY)
        {
            Console.WriteLine("[{0}] {1}",

```

```

        System.AppDomain.CurrentDomain.FriendlyName,
        "Shape constructor");
    upperLeft = new Point(upperLeftX, upperLeftY);
}

public void ShowUpperLeft()
{
    Console.WriteLine("[{0}] Upper left: {1}, {2}",
        System.AppDomain.CurrentDomain.FriendlyName,
        upperLeft.X, upperLeft.Y);
}

public Point GetUpperLeft()
{
    return upperLeft;
}

private Point upperLeft;
}

public class Testter
{
    public static void Main()
    {
        Console.WriteLine("[{0}] {1}",
            System.AppDomain.CurrentDomain.FriendlyName,
            "Entered Main");

        // tạo một app domain mới
        AppDomain ad2 = System.AppDomain.CreateDomain(
            "Shape Domain");

        // Assembly a = Assembly.LoadFrom("MarshalAppDomain.exe");
        // Object theShape = a.CreateInstance("Shape");
        // hiển lộ một đối tượng Shape
        ObjectHandle oh = ad2.CreateInstance(
            "MarshalAppDomain", // tên assembly
            "MarshalAppDomain.Shape", // tên kiểu với namespace
            false, // ignore case
            System.Reflection.BindingFlags.CreateInstance, // flag
            null, // binder
            new object[] {3, 5}, // args
            null, // culture
            null, // activation attribute
            null ); // security attribute

        Shape s1 = (Shape) oh.Unwrap();

        s1.ShowUpperLeft(); // yêu cầu đối tượng hiển thị

        // lấy một bản sao local? proxy? và gán trị mới
        Point localPoint = s1.GetUpperLeft();
        localPoint.X = 500;
        localPoint.Y = 600;

        // cho hiển thị trị của đối tượng Point local
        Console.WriteLine("[{0}] localPoint: {1}, {2}",

```

```

        System.AppDomain.CurrentDomain.FriendlyName,
        localPoint.X, localPoint.Y);
    sl.ShowUpperLeft(); // cho hiển thị trị một lần nữa
    Console.ReadLine();
}
}
}

```

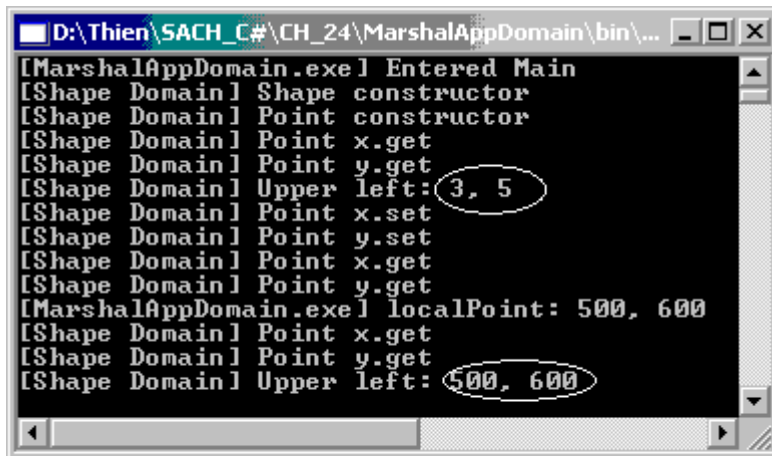
Hình 5-03 (trang 11) cho thấy kết xuất của chương trình thí dụ 5-02 kể trên. Cũng thí dụ trên, bây giờ bạn comment out attribute [Serializable] và uncomment base class như sau:

```

// [Serializable]
public class Point: MarshalByRefObject

```

rồi cho chạy lại chương trình, và hình 5-04 cho thấy kết xuất:



**Hình 5-04: Kết xuất thí dụ 5-02 với lớp Point bị marshal theo reference.**

Lần này bạn nhận một proxy đối với đối tượng Point và các thuộc tính được đặt để thông qua proxy trên biến thành viên nguyên thủy của Point. Do đó những thay đổi được phản ánh liền lên bản thân Shape. Bạn so sánh hai hình 5-03 và 5-04.

## 5.2 Phạm trù (context)

App domain lại được chia nhỏ thành *context* (phạm trù). Context được xem như là ranh giới trong lòng những đối tượng cùng chia sẻ sử dụng các qui tắc (rule). Việc sử dụng các qui tắc này bao gồm việc đồng bộ hoá các phiên giao dịch. (Xem chương 6, “Mạch trình và Đồng bộ hoá”).

## 5.2.1 Các đối tượng contex-bound và context-agile

Các đối tượng có thể hoặc là *context-bound* hoặc *context-agile*. Nếu các đối tượng thuộc loại context-bound (được gắn kết với phạm trù) thì chúng hiện hữu ngay trong phạm trù, và khi muốn tương tác với chúng thông điệp phải được marshal. Còn khi chúng thuộc loại context-agile (nhánh nhảy theo phạm trù) thì chúng hành động ngay trong lòng phạm trù của đối tượng phía triệu gọi; nghĩa là các hàm hành sự của chúng sẽ thi hành ngay trong phạm trù của đối tượng phía triệu gọi và không nhất thiết phải được marshal.

Giả sử bạn có một đối tượng A tương tác với căn cứ dữ liệu và như thế đối tượng A được đánh dấu là chịu hỗ trợ các phiên giao dịch. Điều này tạo ra một phạm trù. Các hàm hành sự triệu gọi A sẽ xảy ra trong lòng phạm trù bảo vệ mà các phiên giao dịch tự trang bị. Đối tượng A có thể quyết định cho trở lộn lui lại (rollback) phiên giao dịch, và những hành động đã được thực hiện từ lần trình duyệt<sup>15</sup> (commit) chót sẽ bị hoá giải (undone).

Giả sử, bạn có một đối tượng B khác thuộc loại context-agile. Bây giờ ta giả định là đối tượng A chuyển một qui chiếu căn cứ dữ liệu cho đối tượng B, rồi sau đó triệu gọi các hàm trên B. Có thể A và B nằm trong một mối liên hệ call-back, theo đấy B sẽ làm vài việc chi đó và sau đó gọi lại A để trao lại kết quả xử lý. Vì B là context-agile, hàm hành sự của B được thi hành trong phạm trù của phía đối tượng triệu gọi là A; như vậy nó sẽ hưởng sự bảo vệ giao dịch của đối tượng A. Những thay đổi mà B thực hiện đối với căn cứ dữ liệu sẽ bị hoá giải nếu đối tượng A rollback giao dịch, vì các hàm hành sự của B được thi hành trong phạm trù của phía triệu gọi.

Như vậy B phải là context-agile hay là context-bound? Trong trường hợp ta vừa quan sát, B hoạt động tốt khi đang là context-agile. Bây giờ ta giả định một lớp hiện hữu nữa: lớp C. Lớp C không có các phiên giao dịch, và nó sẽ triệu gọi hàm hành sự của B lo thay đổi gì đó trên căn cứ dữ liệu. Bây giờ A cố rollback, nhưng rất tiếc phần việc mà B làm cho C lại nằm trong phạm trù C và như vậy không nhận sự hỗ trợ giao dịch của A, và như vậy công việc mà B đã làm giùm cho C không thể bị hoá giải.

Nếu B được đánh dấu context-bound, và khi A tạo ra B, B sẽ kế thừa phạm trù của A. Trong trường hợp này, khi C cho triệu gọi một hàm hành sự trên B nó phải được marshal xuyên ranh giới phạm trù, nhưng khi B cho thi hành hàm hành sự nó phải ở trong phạm trù của phiên giao diện của A. Như vậy là tốt hơn.

Điều này sẽ chạy nếu B là context-bound nhưng không có attribute. Lẽ dĩ nhiên là B có thể có riêng context attribute riêng cho mình, và những attribute này có thể ép B vào trong một phạm trù khác với A. Thí dụ, B có thể có một transaction attribute được đánh

---

<sup>15</sup> Ở đây, cụm từ “trình duyệt” đừng nhầm với browser (mà người ta dịch là trình duyệt) có nghĩa là “trình lên để duyệt xét” (commit).

dấu là **RequireNew**. Trong trường hợp này, khi B được tạo nó sẽ nhận một phạm trù mới, như vậy sẽ không ở trong phạm trù của A. Như vậy, khi A cho rollback, công việc B đã thực hiện sẽ không được rollback. Có thể bạn cho đánh dấu **RequireNew** đối với B vì B là một hàm kiểm toán (audit function). Khi A thực hiện một hành động trên căn cứ dữ liệu nó thông báo B lo nhật tu một audit trail (theo dõi kiểm toán). Bạn không muốn công việc của B bị hoá giải khi A rollback phiên giao dịch. Bạn muốn B nằm trong phạm trù giao dịch riêng của một và chỉ rollback khi B phạm sai lầm, chứ không phải của A.

Như vậy, một đối tượng sẽ có 3 lựa chọn: lựa chọn thứ nhất là context-agile, hoạt động trong phạm trù của đối tượng phía triệu gọi. Lựa chọn thứ hai là context-bound (được thực hiện bằng cách dẫn xuất từ **ContextBoundObject**) nhưng lại không có attribute và như thế sẽ hoạt động trong phạm trù của phía tạo dựng. Lựa chọn thứ ba là context-bound với context attribute, và như thế chỉ sẽ hoạt động trong phạm trù nào khớp với context attribute.

Việc bạn quyết định lựa chọn nào tùy thuộc việc đối tượng của bạn sẽ được sử dụng thế nào. Nếu đối tượng của bạn là một đối tượng đơn giản, như một máy tính bỏ túi (calculator) chẳng hạn, không cần đến đồng bộ hoá hoặc phiên giao dịch hoặc bất cứ hỗ trợ phạm trù nào, thì nên chọn context-agile là hiệu quả nhất. Nếu đối tượng của bạn phải sử dụng phạm trù của đối tượng tạo ra nó, thì đối tượng này phải là context-bound không mang attribute. Cuối cùng, nếu đối tượng của bạn có những yêu cầu phạm trù riêng, bạn phải cho đối tượng này là context-bound với attribute thích ứng.

## 5.2.2 Cho marshal xuyên biên giới phạm trù

Ta sẽ không cần đến proxy khi muốn truy xuất các đối tượng context-agile trong lòng một app domain đơn độc. Khi một đối tượng nằm trong một phạm trù muốn truy xuất một đối tượng context-bound nằm trong phạm trù thứ hai, nó phải truy xuất thông qua một proxy, và vào lúc đó, hai cơ chế phạm trù phải được tuân thủ. Nghĩa là theo chiều hướng này một phạm trù sẽ tạo ra ranh giới; cơ chế sẽ được tuân thủ ở ngay ranh giới phân chia hai phạm trù.

Thí dụ, khi bạn đánh dấu một đối tượng là context-bound thông qua attribute **System.Runtime.Remoting.Synchronization**, bạn cho biết là bạn muốn hệ thống quản lý việc đồng bộ hoá đối với đối tượng này. Tất cả các đối tượng nằm ngoài phạm trù này phải vượt qua phạm trù ranh giới để đến với một trong những đối tượng này, và vào lúc đó cơ chế đồng bộ hoá sẽ được áp dụng.

Các đối tượng sẽ được marshal khác nhau xuyên ranh giới phạm trù tùy vào việc các đối tượng này được tạo ra thế nào:

- Các đối tượng điển hình thường không được marshal chi cả; trong lòng app domain các đối tượng này thường thuộc context-agile.
- Các đối tượng được đánh dấu bởi attribute **Serializable** thường được marshal theo tri xuyên app domain và thuộc loại context-agile.
- Các đối tượng được dẫn xuất từ **MarshalByRefObject** sẽ được marshal theo qui chiếu xuyên app domain và thuộc loại context-agile.
- Các đối tượng được dẫn xuất từ **ContextBoundObject** sẽ được marshal theo qui chiếu xuyên app domain cũng như theo qui chiếu xuyên ranh giới phạm trù.

## 5.3 Remoting<sup>16</sup>

Ngoài việc được marshal xuyên ranh giới app domain và ranh giới phạm trù, các đối tượng có thể được marshal xuyên ranh giới các process, kể cả xuyên ranh giới các máy tính. Khi một đối tượng được marshal, hoặc theo tri hoặc theo qui chiếu (thông qua một proxy), xuyên process hoặc xuyên máy tính, thì nó được gọi là *remoting*.

### 5.3.1 Tìm hiểu kiểu dữ liệu đối tượng Server

Có hai kiểu dữ liệu của các đối tượng server được hỗ trợ bởi .NET đối với remoting: *well-known* và *client-activated*. Việc thông thương liên lạc đối với các đối tượng quá quen thuộc (well-known) sẽ được thiết lập mỗi lần một thông điệp được gửi đi bởi khách hàng. Sẽ không có việc kết nối lâu dài cố định với các đối tượng well-known, như với các đối tượng được khởi động bởi khách hàng (client-activated).

Các đối tượng well-known lại được chia thành hai nhóm: *singleton* và *single-call*. Với một đối tượng well-known singleton, tất cả các thông điệp gửi cho đối tượng đến từ mọi khách hàng sẽ được chuyển cho (dispatched) một đối tượng đơn độc chạy trên server. Đối tượng sẽ được tạo ra khi server được khởi động và nằm đó chờ cung cấp dịch vụ cho bất cứ khách hàng nào tiếp xúc được đối tượng. Các đối tượng well-known phải có một hàm constructor *không thông số*.

Với một đối tượng well-known single-call, thì mỗi thông điệp mới từ khách hàng sẽ được thụ lý bởi một đối tượng mới. Việc này rất tiện lợi đối với những server farm (trang trại server) theo đây một loạt thông điệp từ một khách hàng sẽ được thụ lý đến phiên bởi những máy tính khác nhau tùy thuộc vào sự cân bằng tải (load balancing).

---

<sup>16</sup> Remoting có nghĩa là việc gì dính dáng đến hoạt động từ xa. Tạm thời chúng tôi không dịch.

Còn các đối tượng client-activated điển hình được sử dụng bởi lập trình viên nào tạo những server tận tụy (dedicated server) được tạo ra để cung cấp dịch vụ đối với một khách hàng mà họ viết cho. Theo kịch bản này, client và server tạo một kết nối và duy trì kết nối này cho tới khi nhu cầu của khách hàng được thỏa mãn.

## 5.3.2 Khai báo một Server với một Interface

Cách hay nhất để hiểu remoting là đi thẳng vào một thí dụ. Bạn thử tạo một máy tính bỏ túi mang 4 chức năng (cộng trừ nhân chia) trên web service (xem chương 9, "Lập trình Web Service") thì công một giao diện như theo thí dụ 5-03.

### *Thí dụ 5-03: Giao diện Calculator*

```
using System;

namespace CalcServer
{
    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);
    }
}
```

Nếu bạn chạy trên Visual Studio .NET, bạn cho tạo một dự án mới kiểu C# Class Library, cho mang tên **ICalc.cs**, rồi cho Build thành **ICalc.dll**. Nếu bạn sử dụng **csc.exe** trên command line thì bạn cho ghi đoạn mã trên lên Notepad, cho cất trữ dưới tên **ICalc.cs** rồi cho biên dịch tập tin trên ở command line bằng cách gõ vào

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

Có rất nhiều lợi điểm khi cho thi công một server thông qua một giao diện. Nếu bạn thi công calculator như là một lớp, khách hàng phải kết nối với lớp này để có thể khai báo những thể hiện trên client. Điều này giảm đi rất nhiều những lợi điểm của remoting vì những thay đổi trên server đòi hỏi việc định nghĩa lớp phải được nhật tu trên client. Nói cách khác, client và server phải được gắn kết (lệ thuộc) với nhau một cách quá chặt chẽ, như vậy không tốt. Chính giao diện giúp giảm bớt sự gắn kết quá chặt này. Thật ra, bạn có thể nhật tu việc thi công này trên server, miễn là server làm tròn khế ước mà giao diện đã đặt ra, còn phía client thì khỏi phải thay đổi gì cả.

## 5.3.3 Xây dựng một Server

Muốn xây dựng một server được dùng trong thí dụ này, bạn cho tạo **CalcServer.cs** trên một dự án mới kiểu **C# Console Application** rồi cho **Build**. Hoặc bạn cũng có thể gõ đoạn mã vào Notepad, cho cất trữ dưới dạng tập tin **CalcServer.cs**, rồi gõ vào lệnh sau đây trên command line:

```
csc CalcServer.cs /t:exe
```

Lớp **Calculator** sẽ thi công giao diện **ICalc**. Nó được dẫn xuất từ **MarshalByRefObject** do đó nó sẽ thông qua một proxy của calculator đối với ứng dụng khách hàng:

```
public class Calculator: MarshalByRefObject, Icalc
```

Việc thi công đòi hỏi nhiều hơn là một hàm constructor và những hàm hành sự đơn giản thi công 4 chức năng. Trong thí dụ này, bạn đưa phần lô gic đối với server vào hàm **Main()** của **CalcServer.cs**.

Công việc đầu tiên của bạn là tạo một channel. Sử dụng HTTP như là phương tiện chuyển tải vì nó đơn giản và không đòi hỏi một kết nối TCP/IP. Bạn có thể dùng kiểu dữ liệu **HTTPChannel** do .NET cung cấp:

```
HttpChannel chan = new HttpChannel(65100);
```

Bạn để ý là bạn đăng ký kênh lên cổng TCP/IP 65100. Chương 1, "Xuất nhập dữ liệu & Sản sinh hàng loạt đối tượng" đã đề cập đến số cổng. Tiếp theo, bạn cho đăng ký kênh với lớp **ChannelServices** sử dụng hàm static **RegisterChannel**:

```
ChannelServices.RegisterChannel(chan);
```

Bước này báo cho .NET biết bạn sẽ cung cấp HTTP services đối với cổng 65100, giống như IIS đã làm đối với cổng 80. Vì bạn đã đăng ký một HTTP channel và không cung cấp một formatter riêng của mình, nên các triệu gọi hàm sẽ dùng SOAP như là formatter mặc nhiên.

Bây giờ, bạn sẵn sàng yêu cầu lớp **RemotingConfiguration** cho đăng ký đối tượng well-known của mình. Bạn phải trao kiểu dữ liệu cho đối tượng bạn muốn đăng ký kèm theo *endpoint*. *Endpoint* là một tên mà **RemotingConfiguration** sẽ gắn liền với kiểu dữ liệu của bạn. Nó hoàn tất địa chỉ. Nếu địa chỉ IP nhận diện máy tính và cổng nhận diện kênh thì endpoint nhận diện ứng dụng hiện hành sẽ cung cấp dịch vụ. Muốn lấy kiểu dữ liệu của đối tượng bạn có thể triệu gọi hàm static **GetType()** của lớp **Type**, trả về một đối tượng **Type**. Bạn trao qua tên đầy đủ của đối tượng mà bạn muốn có:

```
Type calcType = Type.GetType("CalcServer.Calculator");
```



Ngoài ra, bạn trao qua kiểu dữ liệu enum **WellKnownObjectMode** cho biết bạn đăng ký như là **Singleton** hoặc **SingleCall**:

```
RemotingConfiguration.RegisterWellKnownServiceType(  
    calcType, "theEndPoint", WellKnownObjectMode.Singleton);
```

Việc triệu gọi hàm **RegisterWellKnownServiceType** không đưa một byte nào lên mạng mà chỉ là dùng reflection để xây dựng một proxy đối với đối tượng của bạn.

Thí dụ 5-04 cho thấy toàn bộ đoạn mã nguồn:

### ***Thí dụ 5-04: Calculator Server***

```
using System;  
using System.Runtime.Remoting;  
using System.Runtime.Remoting.Channels;  
using System.Runtime.Remoting.Channels.Http;  
namespace CalcServer  
{  
    // thi công lớp Calculator  
  
    public class Calculator: MarshalByRefObject, ICalc  
    {  
        public Calculator()  
        {  
            Console.WriteLine("Calculator constructor");  
        }  
        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 Div(double x, double y)  
        {  
            Console.WriteLine("Div {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 class ServerTest
{
    public static void Main()
    {
        // tạo một channel và cho đăng ký
        HttpChannel chan = new HttpChannel(65100);
        ChannelServices.RegisterChannel(chan);

        Type calcType = Type.GetType("CalcServer.Calculator");

        // đăng ký kiểu well-known và yêu cầu server kết nối
        // về type tại "theEndPoint"
        RemotingConfiguration.RegisterWellKnownServiceType(
            calcType, "theEndPoint", WellKnownObjectMode.Singleton);

        Console.WriteLine("Press [enter] to exit...");
        Console.ReadLine();
    }
}

```

Khi bạn cho chạy chương trình này, nó in ra một thông điệp “ngán ngẩm” như sau (hình 5-05):



Hình 5-05: Kết xuất thí dụ 5-04

### 5.3.4 Xây dựng một Client

Client cũng phải đăng ký một kênh, nhưng vì bạn không lắng nghe trên kênh này, bạn có thể dùng channel 0:

```

HttpChannel chan = new HttpChannel(0);
ChannelServices.RegisterChannel(chan);

```

Bây giờ khách hàng chỉ cần kết nối thông qua các remoting service, trao qua một đối tượng **Type** tượng trưng kiểu dữ liệu đối tượng nó cần đến (trong trường hợp này là giao diện **ICalc**), và URI (Uniform Resource Identifier) của lớp thi công:

```

MarshalByRefObject obj = (MarshalByRefObject)
    RemotingServices.Connect(typeof(CalcServer.ICalc),
        "http://localhost:65100/theEndPoint");

```

Trong trường hợp này, ta giả định server chạy trên máy cục bộ của bạn, do đó URI là *http://localhost*, theo sau bởi cổng của server, 65100, theo sau bởi endpoint bạn khai báo trên server, **theEndPoint**.

Remoting service phải trả về một đối tượng tượng trưng cho giao diện mà bạn yêu cầu. Sau đó bạn có thể ép kiểu đối tượng này về giao diện và bắt đầu sử dụng nó. Vì remoting không thể được bảo đảm (mạng có thể suy sụp, máy chủ có thể không sẵn sàng, v.v..), bạn phải cho bao bởi khối try/catch:

```
try
{
    CalcServer.ICalc calc = obj as CalcServer.ICalc;
    // sử dụng interface để gọi các hàm
    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);
}
```

Bây giờ, bạn có một proxy của Calculator hoạt động trên server, nhưng không dùng được trên client, xuyên process boundary và nếu bạn muốn xuyên machine boundary. Thí dụ 5-05 cho thấy toàn bộ client

### ***Thí dụ 5-05: Remoting Calculator Client***

```
using System;
using System.Runtime.Remoting;
using System.Runtime.Remoting.Channels;
using System.Runtime.Remoting.Channels.Http;

namespace CalcClient
{
    public class CalcClient
    {
        public static void Main()
        {
            int[] myIntArray = new int[3];
            Console.WriteLine("Thien, come here I need you...");
            // Tạo một kênh HTTP và cho đăng ký vào cổng 0
            HttpChannel chan = new HttpChannel(0);
            ChannelServices.RegisterChannel(chan);

            // đi lấy cho ta đối tượng xuyên http channel
            MarshalByRefObject obj = (MarshalByRefObject)
                RemotingServices.Connect (typeof(CalcServer.ICalc),
                    "http://localhost:65100/theEndPoint");

            try
            {
                CalcServer.ICalc calc = obj as CalcServer.ICalc;
            }
        }
    }
}
```

```

        // sử dụng interface để gọi các hàm
        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);

        // in ra kết quả
        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);
        Console.ReadLine();
    }
}
}
}

```

Theo nguyên tắc, bạn thử cho chạy chương trình này, kết xuất sẽ như sau:

```

Kết xuất trên phía client...
Thien, come here I need you...
3+4 = 7
3-4 = -1
3*4 = 12
3/4 = 0.75

```

```

Kết xuất trên phía server...
Calculator constructor
Press [enter] to exit...
Add 3 + 4
Sub 3 - 4
Mult 3 * 4
Div 3 / 4

```

Server bắt đầu khởi động và chờ người sử dụng ấn <Enter> cho biết là có thể đóng lại. Phía client sẽ bắt đầu và cho hiển thị một thông điệp trên console. Client liên triệu gọi hàm mỗi một trong 4 phép tính. Bạn thấy server sẽ in ra thông điệp khi mỗi phép tính được triệu gọi, rồi kết quả sẽ được in ở phía client.

Bây giờ bạn đã biết lập trình cho chạy trên server và cung cấp dịch vụ cho khách hàng.

## 5.3.5 Sử dụng SingleCall

Muốn thấy khác biệt giữa **SingleCall** và **Singleton**, bạn cho thay đổi dòng lệnh trong hàm **Main()**. Sau đây là dòng lệnh hiện hữu:

```
Type calcType = Type.GetType("CalcServer.Calculator");
RemotingConfiguration.RegisterWellKnownServiceType(
    calcType, "theEndPoint", WellKnownObjectMode.Singleton);
```

bạn cho đổi thành:

```
RemotingConfiguration.RegisterWellKnownServiceType(
    calcType, "theEndPoint", WellKnownObjectMode.SingleCall);
```

Rồi cho chạy lại chương trình, kết xuất sẽ như sau:

```
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
```

## 5.3.6 Tìm hiểu RegisterWellKnownServiceType

Khi bạn triệu gọi hàm **RegisterWellKnownServiceType()** đối với server, hiện việc gì đã xảy ra?. Bạn nhớ cho là bạn đã tạo một đối tượng **Type** đối với lớp **Calculator**:

```
Type calcType = Type.GetType("CalcServer.Calculator");
```

Rồi sau đó bạn cho triệu gọi hàm **RegisterWellKnownServiceType()**, trao qua đối tượng **Type** này kèm theo endpoint và enum **Singleton**. Việc này cho CLR biết là cho hiển lộ Calculator và sau đó là gắn liền nó với một endpoint.

Muốn tự mình làm việc này, bạn cần thay đổi thí dụ 5-04, thay đổi hàm **Main()** cho hiển lộ một đối tượng **Calculator**, rồi trao đối tượng **Calculator** cho hàm hành sự **Marshal()** của **RemotingServices** với endpoint mà bạn muốn gắn liền với thể hiện **Calculator** này. Thí dụ 5-06 cho thấy Main() được thay đổi và kết xuất cũng tương tự như với thí dụ 5-04:

### Thí dụ 5-06: Hiển lộ bằng tay và gắn liền Calculator với endpoint

```
public static void Main()
{
    // tạo một channel và cho đăng ký
    HttpChannel chan = new HttpChannel(65100);
    ChannelServices.RegisterChannel(chan);

    Calculator calculator = new Calculator();
    RemotingServices.Marshal(calculator, "theEndPoint");

    Console.WriteLine("Press [enter] to exit...");
    Console.ReadLine();
}
```

Tác dụng rõ ràng là bạn đã hiển lộ một đối tượng Calculator, và cho gắn liền một proxy dùng truyền từ xa với endpoint bạn khai báo.

## 5.3.7 Tìm hiểu Endpoints

Việc gì xảy ra khi bạn đăng ký endpoint này? Rõ ràng là server cho gắn liền endpoint này với đối tượng bạn tạo ra, và khi khách hàng kết nối, endpoint này được dùng như là một chỉ mục trên một bảng như vậy server có thể cung cấp một proxy đối với đúng đối tượng (trong trường hợp này là Calculator).

Nếu bạn không cung cấp một endpoint để khách hàng có thể nói chuyện với đối tượng, thay vào đó bạn có thể viết ra tất cả các thông tin liên quan đến đối tượng Calculator lên một tập tin và trao tập tin vật lý này cho khách hàng. Thí dụ, bạn có thể gửi cho thẳng bạn qua email, và nó có thể nạp lên máy cục bộ của nó.

Khách hàng có thể deserialize đối tượng và tái tạo một proxy để rồi sau đó dùng truy cập đối tượng Calculator trên server của bạn.

Để biết làm thế nào bạn có thể triệu gọi một đối tượng không biết đến endpoint, bạn cho thay đổi hàm **Main()** của thí dụ 5-04 một lần nữa. Lần này, thay vì triệu gọi hàm hành sự **Marshal()** kèm theo endpoint, bạn chỉ cần trao qua đối tượng:

```
ObjRef objRef = RemoteServices.Marshal(calculator);
```

**Marshal()** sẽ trả về một đối tượng **ObjRef**. Đối tượng **ObjRef** thường cất trữ tất cả các thông tin cần thiết để khởi động và liên lạc với một đối tượng nằm ở xa. Khi bạn cung cấp một endpoint, thì phía server sẽ tạo một cái bảng cho gắn liền endpoint với một objRef như vậy server sẽ tạo ra một proxy khi khách hàng cần đến. **ObjRef** chứa đựng tất cả các thông tin mà khách hàng cần đến để xây dựng một proxy, và bản thân **ObjRef** là serializable.

Bạn cho mở một file stream để viết lên một tập tin mới và tạo ra một đối tượng SOAP formatter. Bạn có thể serialize **ObjRef** đối với tập tin này bằng cách triệu gọi hàm **Serialize()** trên formatter, trao qua file stream và **ObjRef** mà bạn nhận về từ **Marshal()**.

Thí dụ 5-07 sau đây là toàn bộ hàm Main() bị thay đổi

**Thí dụ 5-07: Marshalling một đối tượng không cần đến well-know endpoint)**

```
public static void Main()
{
    // tạo một channel và cho đăng ký
    HttpChannel chan = new HttpChannel(65100);
    ChannelServices.RegisterChannel(chan);

    // tạo một thẻ hiện đối tượng Calculator và
    // trực tiếp triệu gọi hàm Marshal
    Calculator calculator = new Calculator();
    ObjRef objRef = RemoteServices.Marshal(calculator);
    FileStream fileStream = new FileStream(
        "calculatorSoap.txt", FileMode.Create);
    SoapFormatter soapFormatter = new SoapFormatter();
    soapFormatter.Serialize(fileStream, objRef);

    fileStream.Close();

    Console.WriteLine("Exported to CalculatorSoap.txt.
        Press [enter] to exit...");
    Console.ReadLine();
}
```

Khi bạn cho chạy server, nó viết tập tin *calculatorSoap.txt* lên đĩa. Server sẽ chờ khách hàng kết nối và có thể phải chờ lâu. Bạn có thể lấy tập tin này đem đến khách hàng và tái tạo trên máy tính của khách hàng. Muốn thế, một lần nữa bạn tạo một channel và cho đăng ký kênh này. Tuy nhiên, lần này, bạn cho mở một đối tượng **fileStream** đối với tập tin bạn vừa chép từ server.

```
FileStream fileStream = new FileStream(
    "calculatorSoap.txt", FileMode.Open);
```

Sau đó, bạn cho hiển lộ một đối tượng **SoapFormatter** và triệu gọi hàm **Deserialize()** đối với formatter trao qua cho hàm tên tập tin rồi nhận về một **ObjRef**:

```
SoapFormatter soapFormatter = new SoapFormatter();
try
{
    ObjRef objRef = (ObjRef) soapFormatter.Deserialize(fileStream);
```

Bạn đã sẵn sàng unmarshal **ObjRef** này, nhận lại một qui chiếu **ICalc**:

```
ICalc calc = (ICalc) RemotingServices.Unmarshal(objRef);
```

Bây giờ, bạn hoàn toàn tự do triệu gọi các hàm trên server thông qua **ICalc**, giao diện này hoạt động như là một proxy đối với đối tượng calculator chạy trên server mà bạn đã mô tả trong tập tin calculatorSoap.txt. Thí dụ 5-08 là toàn bộ đoạn mã thay thế đối với phía client:

***Thí dụ 5-08: Hàm Main() thí dụ 5-05 bị thay đổi (phía client)***

```
using System;
using System.Runtime.Remoting;
using System.Runtime.Remoting.Channels;
using System.Runtime.Remoting.Channels.Http;

namespace CalcClient
{
    public class CalcClient
    {
        public static void Main()
        {
            int[] myIntArray = new int[3];
            Console.WriteLine("Thien, come here I need you...");

            // Tạo một kênh HTTP và cho đăng ký vào cổng 0
            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);

                // sử dụng interface để gọi các hàm
                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);

                // in ra kết quả
                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);  
        }  
    }  
}
```

Khi chương trình client khởi động, tập tin sẽ được đọc từ đĩa và proxy được unmarshal. Đây là một công tác mirror lo marshal và serialize đối tượng trên server. Một khi bạn đã unmarshal proxy, bạn có thể triệu gọi các hàm trên đối tượng Calculator chạy trên server.

## Chương 6

# Mạch Trình và Đồng Bộ Hoá

Mạch trình (thread) là những process tương đối “nhẹ cân” chịu trách nhiệm thực hiện “đa nhiệm” (multitasking) trong lòng một ứng dụng đơn độc. Namespace **System.Threading** cung cấp cho bạn vô số lớp và giao diện cho phép bạn quản lý việc lập trình đa-mạch-trình (multithreading). Tuy nhiên, phần lớn lập trình viên có thể không bao giờ cần quản lý một cách tường minh các mạch trình, vì CLR trừu tượng hóa phần lớn những hỗ trợ mạch trình trong các lớp làm đơn giản hóa công tác mạch trình hóa. Thí dụ, trong chương 1, “Xuất nhập dữ liệu & Sản sinh hàng loạt các đối tượng”, bạn đã thấy làm thế nào tạo những multithreaded stream viết và đọc mà bản thân bạn sẽ không cần nhờ tới việc quản lý các mạch trình.

Phần đầu của chương này sẽ cho bạn thấy làm thế nào tạo, quản lý và “khai tử” (kill) các mạch trình. Cho dù bạn không tự tạo một cách tường minh các mạch trình riêng của mình, bạn cũng muốn bảo đảm là đoạn mã của mình có thể thụ lý nhiều mạch trình nếu nó chạy trong một môi trường đa mạch trình. Điều này đặc biệt quan trọng khi bạn tạo ra những cấu kiện mà các lập trình viên khác có thể dùng đến trong một chương trình chịu hỗ trợ đa mạch trình. Nhất là đối với những nhà triển khai các dịch vụ web. Mặc dù các dịch vụ web (sẽ được đề cập đến ở chương 9, “Lập trình Web Service”) có nhiều attribute của những ứng dụng để bàn, chúng thường chạy trên server, thường vắng bóng một giao diện người sử dụng, và buộc lòng các nhà triển khai ứng dụng phải suy nghĩ những giải pháp phía server sao cho hiệu quả và hoạt động theo đa mạch trình.

Phần thứ hai của chương này sẽ tập trung đề cập đến đồng bộ hóa (synchronization). Khi bạn có nguồn lực hạn chế, có thể bạn cần hạn chế việc truy cập nguồn lực này đối với một mạch trình trong một lúc. Ta có thể so sánh việc này như việc đi vệ sinh trên máy bay. Người ta chỉ cho phép một hành khách vào phòng vệ sinh trong một lúc mà thôi. Bạn thực hiện điều này bằng cách đặt một ổ khoá trên cửa. Khi hành khách muốn sử dụng phòng vệ sinh thì họ phải mở tay cửa phòng; nếu cửa khoá thì hoặc họ bỏ đi làm việc gì đó hoặc nôi đuôi đứng chờ một cách kiên nhẫn. Khi phòng vệ sinh trở nên trống (nghĩa là nguồn lực được giải phóng) thì một hành khách trên hàng nôi đuôi có thể vào phòng vệ sinh, khoá cửa lại.

Có lúc, nhiều mạch trình khác nhau có thể muốn truy cập một nguồn lực trên chương trình của bạn, một tập tin chẳng hạn. Điều quan trọng là phải bảo đảm là chỉ một mạch

trình có thể truy cập nguồn lực trong một lúc mà thôi. Như vậy, bạn phải khoá chặt nguồn lực, cho phép một mạch trình truy cập rồi mở chốt nguồn lực. Việc lập trình các khoá có thể khá tinh vi bảo đảm một phân phối sử dụng công bằng các nguồn lực.

## 6.1 Các mạch trình

Các mạch trình được diễn hình tạo ra khi bạn muốn một chương trình phải làm hai việc ngay lập tức. Thí dụ, giả sử bạn muốn tính số pi (3.141592653589...) dài tới 10 tỉ số lẻ. Bộ xử lý sẽ vui vẻ làm việc này, nhưng trong khi ấy thì trên màn hình khung giao diện sẽ không làm gì được. Vì việc tính pi với số lẻ như thế đòi hỏi phải mất vài triệu năm, có thể bạn lại muốn bộ xử lý cung cấp một nhật tu khi nó tiếp tục làm việc. Ngoài ra, bạn cũng muốn có gắn một nút <Stop> cho phép người sử dụng ngưng công việc bất cứ lúc nào. Muốn chương trình thụ lý việc nút <Stop> bị click, bạn sẽ cần đến một mạch trình thi hành thứ hai.

Một nơi phổ biến sử dụng đến mạch trình là khi bạn phải chờ một tình huống, chẳng hạn nhập liệu của người sử dụng, đọc một tập tin hoặc tiếp nhận dữ liệu đến từ mạng. Giải phóng bộ xử lý chuyển qua làm một việc khác trong khi bạn chờ (chẳng hạn tính trị pi) là một ý kiến tốt, làm cho chương trình của mình có vẻ chạy nhanh hơn.

Tuy nhiên, trong vài trường hợp có thể bạn ghi nhận là threading có thể làm chậm lại công việc. Giả sử ngoài việc tính pi, bạn còn muốn tính loạt số Fibonnaci (1, 1, 2, 3, 5, 8, 13, 21...). Nếu bạn có máy tính với nhiều bộ xử lý thì việc này sẽ chạy nhanh hơn vì mỗi tính toán sẽ có riêng cho mình một mạch trình. Còn nếu máy của bạn chỉ có duy nhất một bộ xử lý (như phần lớn các máy tính để bàn), thì việc tính toán trên nhiều mạch trình chắc chắn sẽ chậm hơn là tính toán cái này xong đến cái kia trong một mạch trình đơn độc, vì bộ xử lý phải liên hồi chuyển bặt qua lại giữa hai mạch trình và phải tốn hao gì đó.

### 6.1.1 Khảo sát namespace System.Threading

Namespace **System.Threading** cung cấp một số kiểu dữ liệu cho phép bạn thực hiện lập trình đa mạch trình. Ngoài việc cung cấp những kiểu dữ liệu tương trưng cho một mạch trình cụ thể nào đó, namespace này còn định nghĩa những lớp có thể quản lý một collection các mạch trình (ThreadPool), một lớp Timer đơn giản (không dựa vào GUI) và vô số lớp cung cấp truy cập được đồng bộ vào dữ liệu được chia sẻ sử dụng. Bảng 6-01 liệt kê một vài lớp cốt lõi của namespace **System.Threading**:

**Bảng 6-01: Các lớp cốt lõi của namespace System.Threading:**

Các lớp thành viên	Mô tả
<b>Interlocked</b>	Lớp này dùng cung cấp truy cập đồng bộ hóa vào dữ liệu được chia sẻ sử dụng (shared data).

<b>Monitor</b>	Lớp này cung cấp việc đồng bộ hóa các đối tượng mạch trình sử dụng khoá chốt (lock) và tín hiệu chờ (wait/signal).
<b>Mutex</b>	Lớp này cung cấp việc đồng bộ hóa sơ đẳng có thể được dùng đối với inter process synchronization.
<b>Thread</b>	Lớp này tượng trưng cho một mạch trình được thi hành trong lòng CLR. Sử dụng lớp này bạn có khả năng bổ sung những mạch trình khác trong cùng AppDomain.
<b>ThreadPool</b>	Lớp này quản lý những mạch trình có liên hệ với nhau trong cùng một process nào đó.
<b>Timer</b>	Cho biết một delegate có thể được triệu gọi vào một lúc được khai báo nào đó. Tác vụ wait được thi hành bởi một mạch trình trong thread pool.
<b>WaitHandle</b>	Lớp này tượng trưng cho tất cả các đối tượng đồng bộ hóa (cho phép multiple wait) vào lúc chạy.
<b>ThreadStart</b>	Lớp này là một delegate chỉ về hàm hành sự nào đó phải được thi hành đầu tiên khi một mạch trình bắt đầu.
<b>TimerCallback</b>	Delegate đối với Timer.
<b>WaitCallback</b>	Lớp này là một delegate định nghĩa hàm hành sự kêu gọi lại (callback) đối với ThreadPool user work item.

### 6.1.1.1 Thử khảo sát lớp Thread

Lớp sở dĩ nổi bật trong tất cả các lớp thuộc namespace **System.Threading** là lớp **Thread**. Lớp này tượng trưng cho một vỏ bọc thiên đối tượng bao quanh một lộ trình thi hành trong lòng một **AppDomain** nào đó. Lớp này định nghĩa một số hàm hành sự (cả static lẫn shared) cho phép bạn tạo mới những mạch trình từ mạch trình hiện hành, cũng như cho treo, ngưng, và khai tử một mạch trình nào đó. Trước tiên, ta thử xem vài thành viên static cốt lõi các lớp Thread như theo bảng 6-02:

**Bảng 6-02: Các thành viên static các lớp Thread**

Các thành viên static	Mô tả
<b>CurrentThread</b>	Thuộc tính read-only này trả về một qui chiếu về mạch trình hiện đang chạy.
<b>GetData()</b>	Đi lấy trị từ slot được khai báo trên mạch trình hiện hành đối với domain hiện hành của mạch trình.
<b>SetData()</b>	Cho đặt để trị lên slot được khai báo trên mạch trình hiện hành đối với domain hiện hành của mạch trình.
<b>GetDomain()</b> <b>GetDomainID()</b>	Đi lấy một qui chiếu về AppDomain hiện hành (hoặc mã nhận diện ID của domain này) mà mạch trình hiện đang chạy trên đây.
<b>Sleep()</b>	Cho ngưng mạch trình hiện hành trong một thời gian nhất định được khai báo.

Ngoài ra, lớp **Thread** cũng hỗ trợ các thành viên cấp đối tượng như theo bảng 6-03:

**Bảng 6-03: Các thành viên cấp đối tượng của lớp Thread**

Các thành viên	Mô tả
<b>IsAlive</b>	Thuộc tính này trả về một trị boolean cho biết liệu xem mạch trình đã khởi động hay chưa.
<b>IsBackground</b>	Đi lấy hoặc đặt để một trị cho biết liệu xem mạch trình là một mạch trình nền hay không.
<b>Name</b>	Thuộc tính này cho phép bạn thiết lập một tên văn bản mang tính thân thiện đối với mạch trình.
<b>Priority</b>	Đi lấy hoặc đặt để ưu tiên của một mạch trình. Có thể được gán một trị lấy từ enumeration <b>ThreadPriority</b> (chẳng hạn <b>Normal</b> , <b>Lowest</b> , <b>Highest</b> , <b>BelowNormal</b> , <b>AboveNormal</b> ).
<b>ThreadState</b>	Đi lấy hoặc đặt để tình trạng của mạch trình. Có thể được gán một trị lấy từ enumeration <b>ThreadState</b> (chẳng hạn <b>Unstarted</b> , <b>Running</b> , <b>WaitSleepJoin</b> , <b>Suspended</b> , <b>SuspendRequested</b> , <b>AbortRequested</b> , <b>Stopped</b> ).
<b>Interrupt()</b>	Cho ngưng chạy mạch trình hiện hành.
<b>Join()</b>	Yêu cầu mạch trình chờ đối với một mạch trình nào đó.
<b>Resume()</b>	Tiếp tục lại đối với một mạch trình bị ngưng chạy.
<b>Start()</b>	Cho bắt đầu thi hành mạch trình được khai báo bởi delegate <b>ThreadStart</b> .
<b>Suspend()</b>	Cho ngưng chạy một mạch trình. Nếu mạch trình đã bị ngưng rồi, một triệu gọi hàm <b>Suspend()</b> sẽ không có tác dụng.

## 6.1.2 Khởi động các mạch trình

Cách đơn giản nhất để tạo một mạch trình là tạo một thể hiện của lớp **Thread**. Hàm constructor của lớp này chỉ nhận một đối mục duy nhất kiểu dữ liệu **delegate**. CLR cung cấp cho bạn lớp delegate **ThreadStart** đặc biệt dành cho mục đích này, chỉ về một hàm hành sự do bạn chỉ định. Như vậy bạn có thể xây dựng một mạch trình và bảo cho mạch trình này “khi khởi động thì cho chạy hàm hành sự này nhé”. Lệnh khai báo delegate **ThreadStart** như sau:

```
public delegate void ThreadStart();
```

Như bạn có thể thấy, hàm hành sự mà bạn gắn liền với delegate sẽ phải không tiếp nhận thông số nào và phải trả về **void**. Do đó, bạn có thể tạo một mạch trình mới như sau:

```
Thread myThread = new Thread(new ThreadStart(myFunc));
```

với **myFunc** phải là một hàm hành sự không tiếp nhận thông số nào và trả về **void**.

Thí dụ, bạn có thể tạo hai mạch trình “thợ”(worker thread), một cho tăng số chẳng hạn:

```
public void Incrementer()
{
    for (int i = 0; i <= 10; i++)
    {
        Console.WriteLine("Incrementer: {0}", i);
    }
}
```

và một giảm số:

```
public void Decrementer()
{
    for (int i = 10; i >=0; i--)
    {
        Console.WriteLine("Decrementer: {0}", i);
    }
}
```

Muốn chạy các hàm trên trên những mạch trình, bạn cho khởi gán mỗi hàm với một đối tượng delegate **ThreadStart**:

```
Thread t1 = new Thread(new ThreadStart(Incrementer));
Thread t2 = new Thread(new ThreadStart(Decrementer));
```

Khởi gán các mạch trình này cũng chưa khởi động chạy. Muốn thế, bạn phải triệu gọi hàm **Start()** thuộc đối tượng **Thread**:

```
t1.Start();
t2.Start();
```

Thí dụ 6-01 là toàn bộ chương trình và hình 6-01 là kết xuất. Vào đầu dự án bạn phải thêm chỉ thị *using System.Threading* để trình biên dịch biết được là có lớp **Thread**. Bạn thấy trên kết xuất bộ xử lý chuyển bật từ t1 qua t2:

### **Thí dụ 6-01: Sử dụng Thread**

```
*****
using System;
using System.Threading;

namespace TestThread
{
    class Tester
    {
        static void Main()
        {
            Tester t = new Tester();
            t.DoTest(); // chạy ngoài static Main
            Console.ReadLine();
        }
    }
}
```

```

public void DoTest()
{
    // tạo một thread đối với Incrementer và
    // một thread khác đối với Decrementer
    Thread t1 = new Thread(new ThreadStart(Incrementer));
    Thread t2 = new Thread(new ThreadStart(Decrementer));

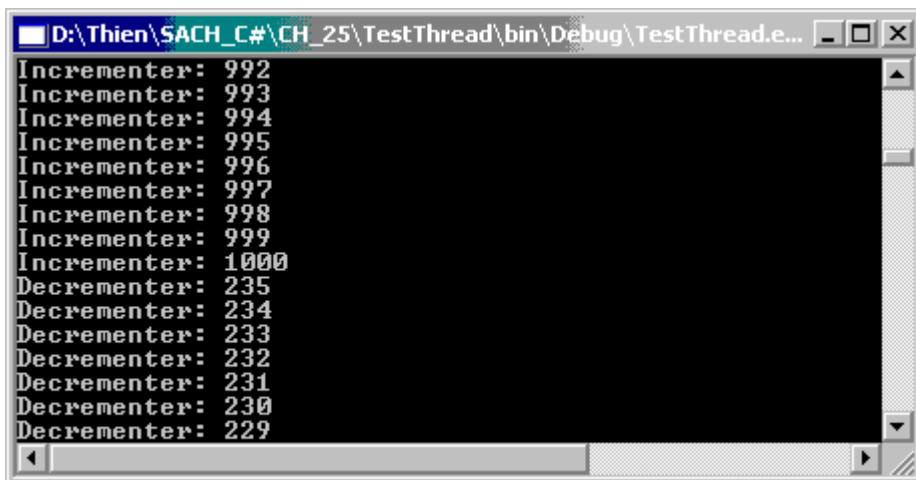
    // khởi động chạy các thread
    t1.Start();
    t2.Start();
}

public void Incrementer()
{
    Console.WriteLine("\n");
    for (int i = 0; i <= 10; i++)
    {
        Console.WriteLine("Incrementer: {0}", i);
    }
}

public void Decrementer()
{
    Console.WriteLine("\n");
    for (int i = 10; i >= 0; i--)
    {
        Console.WriteLine("Decrementer: {0}", i);
    }
}
}
}

```

Hình 6-01 cho thấy kết quả của thí dụ 6-01.



Hình 6-01: Kết xuất thí dụ 6-01

Bộ xử lý cho phép mạch trình thứ nhất chạy tăng con số lên tới số nào đó, rồi chuyển bật qua mạch trình thứ hai bắt đầu đếm từ 1000 xuống trong một lúc, rồi mạch trình thứ nhất lại cho tiếp tục. Thời lượng chuyển bật qua lại do một bộ phận gọi là thread scheduler quản lý tùy thuộc nhiều yếu tố, chẳng hạn tốc độ bộ vi xử lý, nhu cầu đối với bộ xử lý do các chương trình đòi hỏi, v.v..

### 6.1.3 Đặt tên thân thiện cho mạch trình

Một khía cạnh lý thú của lớp **Thread** là nó cho phép gán một tên thân thiện cho mạch trình đang thi hành. Muốn thế, bạn nên tận dụng thuộc tính **Name**. Thí dụ, bạn có thể sửa đổi lại đôi chút diện mạo của thí dụ 6-01 ở trên như sau. Các dòng lệnh in đậm là những thay đổi sử dụng thuộc tính **Name**:

#### *Thí dụ 6-01(B): Sử dụng Thread - phiên bản sử dụng thuộc tính Name*

```
using System;
using System.Threading;

namespace TestThread
{
    class Tester
    {
        static void Main()
        {
            Tester t = new Tester();
            t.DoTest(); // chạy ngoài static Main
            Console.ReadLine();
        }

        public void DoTest()
        {
            // tạo một thread đối với Incrementer và
            // một thread khác đối với Decrementer
            Thread t1 = new Thread(new ThreadStart(Incrementer));
            t1.Name = "Incrementing...";
            Thread t2 = new Thread(new ThreadStart(Decrementer));
            t2.Name = "Decrementing...";
            // khởi động chạy các thread
            t1.Start();
            t2.Start();
        }

        public void Incrementer()
        {
            Console.WriteLine("\n");
            for (int i = 0; i <= 10; i++)
            {
                Console.WriteLine("Value of {0} is {1}: ",
                                Thread.CurrentThread.Name, i);
            }
        }
    }
}
```



```

public void Decrementer()
{
    Console.WriteLine("\n");
    for (int i = 10; i >=0; i--)
    {
        Console.WriteLine("Value of {0} is {1}: ",
                           Thread.CurrentThread.Name, i);
    }
}
}
}
}
*****

```

Kết quả giống như hình 6-01, nhưng phần văn bản có hơi khác: Value of Incrementing... hoặc Value of Decrementing...

## 6.1.4 Ráp lại các mạch trình

Khi bạn bảo một mạch trình ngưng xử lý và chờ mạch trình thứ hai hoàn tất công việc, việc này được xem như là join (ráp lại) mạch trình thứ nhất với mạch trình thứ hai.

Muốn join thread **t1** với thread **t2**, bạn viết:

```
t2.Join();
```

Nếu lệnh này được thi hành trong một hàm hành sự nằm trong mạch trình **t1**, thì **t1** sẽ ngưng chờ cho tới khi **t2** hoàn tất công việc, và thoát ra. Thí dụ, có thể bạn yêu cầu mạch trình mà **Main()** đang thi hành chờ cho tất cả các mạch trình khác kết thúc trước khi nó viết ra thông điệp kết luận. Giả sử bạn tạo ra một collection các mạch trình mang tên **myThreads**. Bạn sẽ cho rảo qua collection này, cho kết mạch trình hiện hành với mỗi một mạch trình trong collection theo phiên:

```

foreach (Thread myThread in myThreads)
{
    myThread.Join();
}
Console.WriteLine("All my threads are done...");

```

Thông điệp cuối cùng **All my threads are done...** chỉ sẽ được in ra khi tất cả các mạch trình hoàn tất công việc. Trong một môi trường sản xuất, có thể cùng lúc bạn khởi động một loạt mạch trình để thực hiện một công tác gì đó (của lớp in, nhật tu màn hình v.v..) và không muốn tiếp tục mạch trình chính cho tới khi các mạch trình “ong thợ” xong việc.

## 6.1.5 Cho treo lại các mạch trình

Đôi lúc, có thể bạn muốn treo mạch trình trong chốc lát. Thí dụ, có thể bạn muốn clock thread (mạch trình đồng hồ) cho ngưng vào khoảng một giây trong lúc trắc nghiệm thời gian hệ thống. Việc này cho phép bạn hiển thị thời gian mới mỗi lúc một giây khỏi phải bắt máy nhọc công phải làm hằng triệu chu kỳ máy.

Lớp **Thread** cung cấp cho bạn một hàm hành sự public static mang tên **Sleep()** dùng cho mục đích này. Hàm này được nạp chồng: một phiên bản nhận vào một thông số **int**, một phiên bản khác lại nhận một đối tượng **timeSpan**. Mỗi **timeSpan** tượng trưng cho một khoảng khắc kéo dài vào khoảng một số milliseconds mà bạn muốn treo mạch trình. Khoảng khắc này được biểu diễn dưới dạng một số nguyên int (nghĩa là 2000 bằng 2000 milliseconds hoặc 2 giây), hoặc dưới dạng một **timeSpan**.

Mặc dù các đối tượng **timeSpan** có thể được đo lường *ticks* (100 nanoseconds), độ tinh xảo của hàm hành sự **Sleep()** được tính theo milliseconds (1.000 nanoseconds).

Muốn bắt mạch trình của bạn ngủ yên trong một giây, bạn có thể viết:

```
Thread.Sleep(1000);
```

Đôi lúc, bạn bảo mạch trình ngủ yên chỉ trong một millisecond. Có thể bạn muốn làm điều này để báo cho thread scheduler rằng bạn muốn mạch trình của mình chuyển giao qua một mạch trình khác cho dù thread scheduler có thể cấp cho mạch trình của bạn nhiều thời gian hơn.

Nếu bạn cho đổi đổi chút thí dụ 6-01, thêm vào lệnh **Thread.Sleep(1)** sau mỗi lệnh **WriteLine()**, thì kết xuất sẽ thay đổi khá lớn:

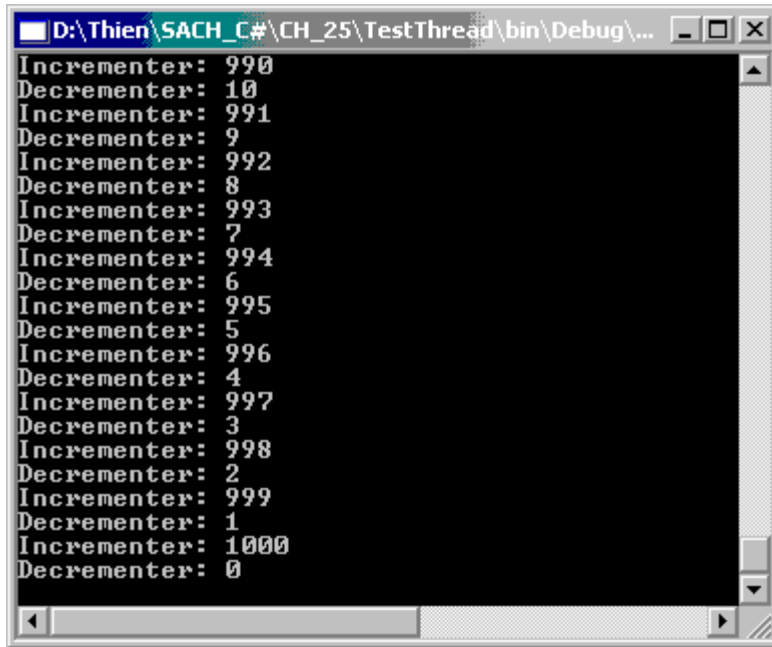
```
for (int i = 0; i <= 1000; i++)
{ Console.WriteLine("Incrementer: {0}", i);
  Thread.Sleep(1);
}
```

Kết quả thay đổi nhỏ này cho ra hình 6-02:

Thay đổi nho nhỏ này đủ mỗi mạch trình có cơ hội chạy một lần một khi mạch trình kia đã in ra một trị. Hình 6-02 phản ánh thay đổi kể trên.

## 6.1.6 Cho khai tử mạch trình

Thông thường mạch trình sẽ “ngủ” luôn một khi đã xong công việc. Tuy nhiên, bạn có thể cho “khai tử” một mạch trình bằng cách triệu gọi hàm **Interrupt()** của mạch trình.



Hình 6-02: Sử dụng đèn Thread.Sleep

Điều này sẽ gây tung ra biệt lệ **ThreadInterruptedException**, mà mạch trình có thể chặn hứng, và do đó cho phép mạch trình có cơ hội giải phóng bất cứ các nguồn lực nào được cấp phát:

```

catch (ThreadInterruptedException)
{ Console.WriteLine("Thread {0} interrupted! Cleaning up...",
    Thread.CurrentThread.Name);
}
  
```

Mạch trình lo xử lý biệt lệ **ThreadInterruptedException** như là một dấu hiệu đã đến lúc phải thoát ra và càng nhanh càng tốt.

Có thể bạn muốn khai tử một mạch trình do phản ứng trước một tình huống chẳng hạn khi người sử dụng ấn nút <Cancel>. Hàm thụ lý tình huống nút Cancel có thể nằm trong mạch trình T1 còn tình huống gây ra Cancel có thể nằm ở mạch trình T2. Trong event handler, bạn có thể triệu gọi hàm **Interrupt** đối với T1:

```
T1.Interrupt();
```

Một biệt lệ sẽ được tung ra trên hàm hành sự đang chạy của T1 mà T1 có thể chặn hứng. Việc này cho T1 có cơ hội giải phóng các nguồn lực và thoát ra nhẹ nhàng.

Trên thí dụ 6-02 dưới đây, 3 mạch trình sẽ được tạo ra và được trữ trên một bản dây các đối tượng **Thread**. Trước khi **Threads** được bắt đầu, thuộc tính **IsBackground** được cho về true. Mỗi mạch trình sẽ được bắt đầu và được đặt tên (nghĩa là Thread1, Thread2, v.v...). Một thông điệp sẽ được hiển thị cho biết mạch trình nào được bắt đầu và mạch trình chính sẽ nằm yên trong 50 milliseconds trước khi bắt đầu mạch trình kế tiếp.

Sau khi tất cả 3 mạch trình được bắt đầu và 50 milliseconds khác đã trôi qua, mạch trình thứ nhất sẽ bị khai tử bằng cách triệu gọi hàm **Interrupt()**. Mạch trình chính sau đó sẽ kết 3 mạch trình đang chạy. Hậu quả của việc này mạch trình chính sẽ không kết thúc khi 3 mạch trình kia chưa xong việc. Khi xong việc, mạch trình chính sẽ in ra thông điệp: All my threads are done. Sau đây, là toàn bộ đoạn mã nguồn của thí dụ 6-02:

### **Thí dụ 6-02: Cho khai tử một mạch trình**

```
*****
using System;
using System.Threading;

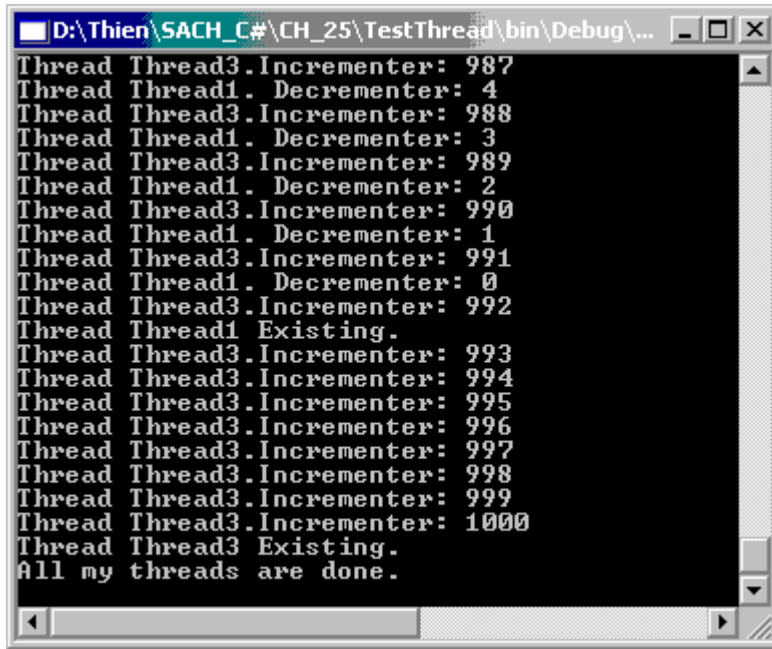
namespace TestThread
{
    class Tester
    {
        static void Main()
        {
            Tester t = new Tester();
            t.DoTest();
            Console.ReadLine();
        }

        public void DoTest()
        {
            // Tạo một bản dây những mạch trình vô danh
            Thread[] myThreads =
                {
                    new Thread(new ThreadStart(Decrementer)),
                    new Thread(new ThreadStart(Incrementer)),
                    new Thread(new ThreadStart(Incrementer))
                };

            // khởi động mỗi 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);
            }

            // sau khi đã khởi động các mạch trình
            // yêu cầu mạch trình số 1 khai tử
            myThreads[1].Interrupt();
        }
    }
}
```





```
Thread Thread3.Incrementer: 987
Thread Thread1. Decrementer: 4
Thread Thread3.Incrementer: 988
Thread Thread1. Decrementer: 3
Thread Thread3.Incrementer: 989
Thread Thread1. Decrementer: 2
Thread Thread3.Incrementer: 990
Thread Thread1. Decrementer: 1
Thread Thread3.Incrementer: 991
Thread Thread1. Decrementer: 0
Thread Thread3.Incrementer: 992
Thread Thread1 Existing.
Thread Thread3.Incrementer: 993
Thread Thread3.Incrementer: 994
Thread Thread3.Incrementer: 995
Thread Thread3.Incrementer: 996
Thread Thread3.Incrementer: 997
Thread Thread3.Incrementer: 998
Thread Thread3.Incrementer: 999
Thread Thread3.Incrementer: 1000
Thread Thread3 Existing.
All my threads are done.
```

Hình 6-03: Kết xuất thí dụ 6-02.

Bạn sẽ thấy là mạch trình 1 bắt đầu và giảm từ 1000 xuống một số nào đó, rồi mạch trình 2 bắt đầu và hai mạch trình này thay phiên nhau trong một lúc thì mạch trình 3 bắt đầu. Tuy nhiên, sau một thời gian ngắn, Thread2 báo cáo là nó bị ngưng và cho biết là thoát ra. Hai mạch trình còn lại tiếp tục cho tới khi xong việc, và thoát ra một cách tự nhiên, và mạch trình chính chấm dứt với thông điệp cuối cùng.

## 6.2 Đồng bộ hóa (synchronization)

Đôi lúc, có thể bạn muốn điều khiển việc truy cập vào một nguồn lực, chẳng hạn các thuộc tính hoặc hàm hành sự của một đối tượng, làm thế nào chỉ một mạch trình được phép thay đổi hoặc sử dụng nguồn lực mà thôi. Đối tượng của bạn cũng giống như phòng vệ sinh trong máy bay, và các mạch trình khác nhau tương tự như hàng nối đuôi hành khách chờ đi vệ sinh. Việc đồng bộ hóa được thể hiện thông qua một cái khoá được thiết lập trên đối tượng, ngăn không cho mạch trình nào đó chui vào khi mạch trình đi trước chưa xong công việc.

Trong mục này, bạn sẽ làm quen với 3 cơ chế đồng bộ hóa mà CLR cung cấp cho bạn: lớp **Interlock**, lệnh C# **lock**, và lớp **Monitor**. Nhưng trước tiên, bạn cần mô phỏng một nguồn lực được chia sẻ sử dụng, chẳng hạn một tập tin hoặc một máy in bằng cách

sử dụng một biến số nguyên đơn giản: **counter**. Thay vì mở một tập tin hoặc truy cập một máy in, bạn tăng counter đối với mỗi mạch trình.

Để bắt đầu, bạn cho khai báo biến thành viên và khởi gán về zero:

```
int counter = 0;
```

Bạn cho thay đổi hàm hành sự Incrementer để tăng biến **counter**:

```
public void Incrementer()
{
    try
    {
        while (counter < 1000)
        {
            int temp = counter;
            temp++; // tăng

            // mô phỏng công việc gì đó trong hàm này
            Thread.Sleep(1);

            // gán trị tăng cho biến counter và hiển thị kết quả
            counter = temp;
            Console.WriteLine("Thread {0}.Incrementer: {1}",
                              Thread.CurrentThread.Name, counter);
        }
    }
}
```

Ở đây, ta muốn mô phỏng công việc có thể phải làm đối với một nguồn lực được kiểm soát. Giống như ta có thể mở một tập tin, xào xáo nội dung tập tin và sau đó là cho đóng tập tin lại, ở đây ta đọc trị của counter vào một biến trung gian, tăng biến trung gian, nằm yên 1 millisecond để mô phỏng một công việc nào đó, rồi sau đó gán trị đã tăng trả về cho counter.

Bài toán ở đây như sau: mạch trình thứ nhất sẽ đọc trị counter (0) rồi gán trị này cho biến trung gian (temp). Sau đó sẽ tăng trị của temp. Trong khi nó làm công việc, thì mạch trình thứ hai sẽ đọc trị của counter (đang còn là zero) rồi gán trị này cho biến trung gian. Mạch trình thứ nhất xong việc thì cho gán trị của temp (1) trả về cho counter và cho hiển thị trị này. Mạch trình thứ hai cũng làm như thế. Như vậy sẽ in ra 1, 1. Trong phiên kế tiếp việc này xảy ra giống như trước. Kết quả là thay vì hai mạch trình đếm 1, 2, 3, 4, ... ta sẽ thấy 1, 1, 2, 2, 3, 3, 4, 4,.. Thí dụ 6-03 sau đây cho thấy toàn bộ mã nguồn và kết xuất:

### ***Thí dụ 6-03: Mô phỏng một nguồn lực được chia sẻ sử dụng***

\*\*\*\*\*

```
using System;
using System.Threading;
```

```
namespace TestThread
{
    class Tester
    {
        private int counter = 0;

        static void Main()
        {
            Tester t = new Tester();
            t.DoTest();
            Console.ReadLine();
        }

        public void DoTest()
        {
            Thread t1 = new Thread(new ThreadStart(Incrementer));
            t1.IsBackground = true;
            t1.Name = "ThreadOne";
            t1.Start();
            Console.WriteLine("Started thread {0}", t1.Name);

            Thread t2 = new Thread(new ThreadStart(Incrementer));
            t2.IsBackground = true;
            t2.Name = "ThreadTwo";
            t2.Start();
            Console.WriteLine("Started thread {0}", t2.Name);
            t1.Join();
            t2.Join();
            Console.WriteLine("All my threads are done.");
        }

        public void Incrementer()
        {
            try
            {
                while (counter < 1000)
                {
                    int temp = counter;
                    temp++; // tăng

                    // mô phỏng công việc gì đó trong hàm này
                    Thread.Sleep(1);

                    // gán trị tăng cho biến counter và hiển thị kết quả
                    counter = temp;
                    Console.WriteLine("Thread {0}.Incrementer: {1}",
                        Thread.CurrentThread.Name, counter);
                }
            }
            catch (ThreadInterruptedException)
            {
                Console.WriteLine("Thread {0} interrupted!
                    Cleaning up...", Thread.CurrentThread.Name);
            }
            finally
            {
                Console.WriteLine("Thread {0} Existing. ",
                    Thread.CurrentThread.Name);
            }
        }
    }
}
```

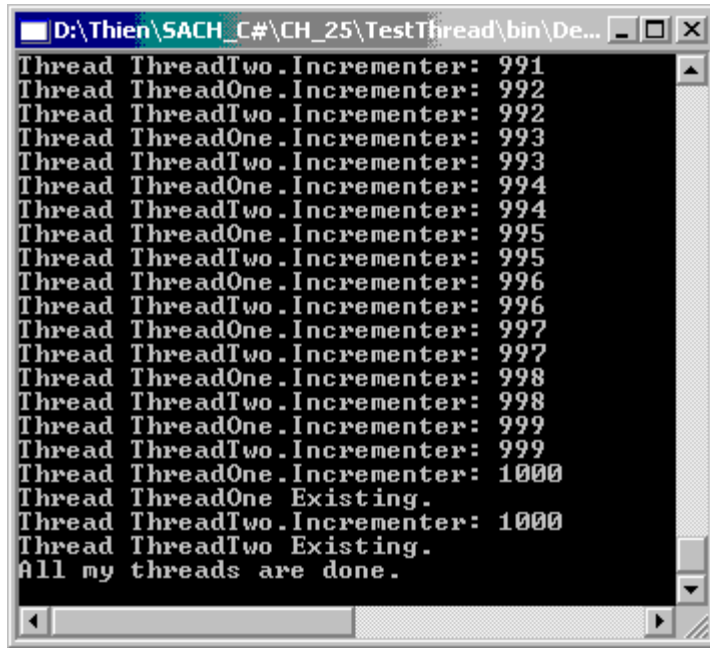


```

*****
    }
}

```

Hình 6-04 cho thấy kết xuất của thí dụ 6-03 kể trên.



Hình 6-04:: Kết xuất Thí dụ 6-03

Giả sử hai mạch trình của bạn truy cập một mẫu tin căn cứ dữ liệu thay vì một biến thành viên. Thí dụ, có thể đoạn mã của bạn là một hệ thống tồn kho của một nhà bán sách. Một khách hàng có thể hỏi xem tập sách *Căn bản lập trình C#* có sẵn hay không. Mạch trình thứ nhất đọc vào yêu cầu và tìm ra có một quyền trong kho. Khách hàng đồng ý mua quyền sách này, do đó mạch trình tiến hành thủ tục bán sách làm hóa đơn và các thông tin thanh toán cũng như kiểm tra hợp lệ liên quan đến địa chỉ của khách hàng.

Trong khi việc kể trên tiến triển, một mạch trình thứ hai cũng hỏi xem quyền sách trên có sẵn hay không. Mạch trình thứ nhất chưa nhật tu mẫu tin tồn kho, do đó quyền sách vẫn còn trong kho, và chưa cho thấy dấu hiệu là đã có người mua rồi. Thế là mạch trình thứ hai tiến hành thủ tục bán sách khi khách hàng đồng ý mua quyền sách này. Trong khi ấy, thì mạch trình thứ nhất xong công việc và cho tồn kho về zero. Mạch trình thứ hai không hề ý thức việc làm của mạch trình thứ nhất nên cũng cho trị tồn kho về zero. Như vậy điều đáng tiếc ở đây là bạn đã bán hai lần cùng một quyền sách duy nhất còn lại trong kho.

Do đó, bạn cần đồng bộ hóa việc truy cập đối tượng **counter** ( hoặc đối với mẫu tin căn cứ dữ liệu, tập tin hoặc máy in, v.v..).

## 6.2.1 Sử dụng System.Threading.Interlocked

CLR cung cấp cho bạn một số cơ chế đồng bộ hóa, bao gồm các công cụ đồng bộ hóa thông dụng chẳng hạn “critical section” (phần hành căng), được gọi là **Locks** trên .NET cũng như các công cụ tinh vi hơn chẳng hạn lớp **Monitor**. Chúng ta sẽ lần lượt đi qua các công cụ này.

Việc tăng và giảm một trị nào đó là một tập tục khá phổ biến trong lập trình, và cần có sự bảo vệ đồng bộ hóa. C# cung cấp một lớp đặc biệt, **Interlocked**, cho mục đích này. Lớp này có hai hàm hành sự: **Increment** và **Decrement**, lớp tăng hoặc giảm một trị nào đó nhưng đồng thời lại lo điều khiển việc đồng bộ hóa.

Bạn cho thay đổi hàm hành sự **Incrementer** trên thí dụ 6-03 lại như sau:

```
*****
public void Incrementer()
{
    try
    {
        while (counter < 1000)
        {
            Interlocked.Increment(ref counter);
            // mô phỏng công việc gì đó trong hàm này
            Thread.Sleep(1);

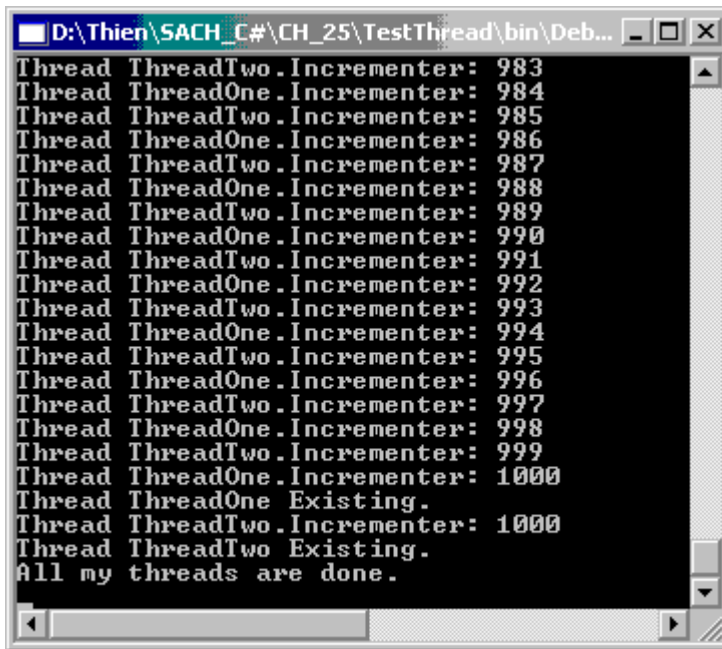
            // gán trị tăng cho biến counter và hiển thị kết quả
            Console.WriteLine("Thread {0}.Incrementer: {1}",
                              Thread.CurrentThread.Name, counter);
        }
    }
    catch (ThreadInterruptedException)
    {
        Console.WriteLine("Thread {0} interrupted!
                          Cleaning up...", Thread.CurrentThread.Name);
    }
    finally
    {
        Console.WriteLine("Thread {0} Existing. ",
                          Thread.CurrentThread.Name);
    }
}
*****
```

Bạn thấy là khối catch và finally không thay đổi.

Hàm **Interlocked.Increment** chờ đợi một thông số duy nhất: một qui chiếu về một số nguyên int. Vì int được trao theo trị, bạn phải sử dụng từ chốt **ref** như đã được mô tả trong chương 5, Tập I, “Lớp và Đối tượng”.

**Bạn để ý:** Hàm hành sự **Increment()** là một hàm được nạp chồng, nên nó có thể qui chiếu về một long thay vì một int, nếu bạn thấy tiện hơn.

Một khi sự thay đổi được tiến hành việc truy cập vào biến thành viên counter sẽ được đồng bộ hóa và kết xuất như theo hình 6-05.



```
D:\Thien\SACH_C#\CH_25\TestThread\bin\Deb...
Thread ThreadTwo.Incrementer: 983
Thread ThreadOne.Incrementer: 984
Thread ThreadTwo.Incrementer: 985
Thread ThreadOne.Incrementer: 986
Thread ThreadTwo.Incrementer: 987
Thread ThreadOne.Incrementer: 988
Thread ThreadTwo.Incrementer: 989
Thread ThreadOne.Incrementer: 990
Thread ThreadTwo.Incrementer: 991
Thread ThreadOne.Incrementer: 992
Thread ThreadTwo.Incrementer: 993
Thread ThreadOne.Incrementer: 994
Thread ThreadTwo.Incrementer: 995
Thread ThreadOne.Incrementer: 996
Thread ThreadTwo.Incrementer: 997
Thread ThreadOne.Incrementer: 998
Thread ThreadTwo.Incrementer: 999
Thread ThreadOne.Incrementer: 1000
Thread ThreadOne Existing.
Thread ThreadTwo.Incrementer: 1000
Thread ThreadTwo Existing.
All my threads are done.
```

Hình 6-05: Sử dụng **Interlocked.Increment**

## 6.2.1 Sử dụng từ chốt C# Locks

Mặc dù đối tượng **Interlocked** rất tốt khi bạn muốn tăng hoặc giảm một trị, nhưng đôi lúc bạn muốn điều khiển việc truy cập các đối tượng khác. Điều ta muốn ở đây là một cơ chế đồng bộ hóa tổng quát hơn. Điều này sẽ được cung cấp bởi đối tượng **Lock**.

Một lock đánh dấu một phân đoạn căng thẳng (critical section) trên đoạn mã của bạn, đồng thời cung cấp việc đồng bộ hóa đối với một đối tượng bạn chỉ định khi lock có hiệu lực. Cú pháp sử dụng một **Lock** là yêu cầu khóa chặt một đối tượng rồi thi hành một câu lệnh hoặc một khối lệnh. Sẽ mở khoá khi ta vào cuối khối lệnh.

C# cung cấp hỗ trợ trực tiếp việc khoá chặt thông qua từ chốt **lock**. Bạn trao qua theo một đối tượng qui chiếu và theo sau từ chốt là một khối lệnh:

```
lock(expression) statement-block
```

Thí dụ, một lần nữa bạn có thể thay đổi hàm **Incrementer** sử dụng một câu lệnh lock như sau:

```
public void Incrementer()
{
    try
    {
        while (counter < 1000)
        {
            lock(this)
            {
                int temp = counter;
                temp++;
                Thread.Sleep(1);
                counter = temp;
            }

            // gán trị tăng cho biến counter và hiển thị kết quả
            Console.WriteLine("Thread {0}.Incrementer: {1}",
                              Thread.CurrentThread.Name, counter);
        }
    }
}
```

Các khối catch và finally không thay đổi. Kết xuất cũng tương tự như với việc sử dụng **Interlocked**.

## 6.2.2 Sử dụng System.Threading.Monitor

Những đối tượng khoá chặt bạn dùng mãi tới đây khá đủ cho phần lớn các nhu cầu. Còn đối với nhiều việc điều khiển các nguồn lực tinh vi hơn thì có lẽ bạn sẽ cần đến một *monitor*. Một monitor cho phép bạn quyết định lúc nào chui vào đồng bộ hóa và lúc nào thì thoát ra khỏi đồng bộ hóa, và nó cho phép bạn chờ đợi một vùng đoạn mã sẽ được giải phóng.

Monitor hoạt động giống như một ổ khoá thông minh đối với một nguồn lực. Khi bạn muốn bắt đầu đồng bộ hóa, bạn triệu gọi hàm **Enter()**, trao qua đối tượng bạn muốn đặt khoá:

```
Monitor.Enter(this);
```

Nếu monitor bị bận, có nghĩa là đối tượng mà monitor đang bảo vệ hiện đang hiệu lực. Bạn có thể làm việc gì đó trong khi chờ monitor rảnh trở lại, rồi thử lại lần nữa. Bạn

cũng có thể chọn rõ **Wait()**, ngưng chạy mạch trình chờ cho tới khi monitor rảnh trở lại. **Wait()** giúp bạn điều khiển trật tự mạch trình.

Giả sử bạn đang nạp xuống và in một bài báo gì đó từ Web. Để cho hiệu quả, bạn muốn cho in theo một mạch trình hậu trường, nhưng bạn muốn bản đây là ít nhất 10 trang được nạp xuống trước khi bắt đầu.

Mạch trình in sẽ chờ cho tới khi mạch trình get-file báo hiệu cho biết là đã đọc đủ tập tin. Bạn không cần phải join mạch trình get-file vì tập tin có thể dài hàng trăm trang. Bạn không muốn chờ cho tới khi nó kết thúc việc nạp xuống nhưng bạn muốn bảo đảm là ít nhất 10 trang được nạp xuống trước khi mạch trình in bắt đầu. Hàm **Wait()** sẽ được dùng cho mục đích này.

Muốn mô phỏng việc này, bạn sẽ viết lại phần Tester và đưa vào hàm hành sự Incrementer cho tới 10. Còn hàm Decrementer sẽ đếm lui xuống zero, và bạn không muốn bắt đầu hàm Decrementer trừ khi giá trị của counter ít nhất bằng 5.

Trên hàm Decrementer, bạn triệu gọi hàm Enter đối với monitor. Sau đó bạn kiểm tra giá trị của counter, và nếu nó nhỏ thua 5, bạn có thể triệu gọi hàm **Wait** đối với monitor.

```
if (counter < 5)
{
    Monitor.Wait(this);
}
```

Việc triệu gọi hàm **Wait()** sẽ giải phóng monitor nhưng lại báo cho CLR biết bạn muốn có lại monitor lần tới khi nó rảnh. Các mạch trình đang chờ sẽ được thông báo cơ may chạy lại nếu mạch trình hiện dịch triệu gọi hàm **Pulse()**:

```
Monitor.Pulse();
```

Hàm **Pulse()** báo cho CLR biết là có thay đổi trong trạng thái có thể giải phóng một mạch trình đang chờ. CLR sẽ theo dõi sự kiện mạch trình trước đó yêu cầu chờ, và các mạch trình sẽ được bảo đảm truy cập theo thứ tự theo đây việc chờ được yêu cầu (nghĩa là việc chờ sẽ được thụ lý theo thứ tự yêu cầu).

Khi một mạch trình xong công việc với monitor, nó có thể đánh dấu cho biết cuối vùng mã được kiểm soát thông qua một triệu gọi hàm **Exit()**:

```
Monitor.Exit(0);
```

Thí dụ 6-04 tiếp tục việc mô phỏng, cung cấp việc đồng bộ hóa truy cập vào một biến **counter** sử dụng một **Monitor**:

### **Thí dụ 6-04: Sử dụng đối tượng Monitor**

\*\*\*\*\*

```
using System;
using System.Threading;

namespace MonitorTest
{
    class Tester
    {
        static void Main()
        {
            Tester t = new Tester();
            t.DoTest();
            Console.ReadLine();
        }

        public void DoTest()
        {
            // Tạo một bản dãy những mạch trình vô danh
            Thread[] myThreads =
            {
                new Thread(new ThreadStart(Decrementer)),
                new Thread(new ThreadStart(Incrementer)),
            };

            // khởi động mỗi 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);
            }

            // chờ cho các mạch trình khác xong việc trước khi tiếp tục
            foreach(Thread myThread in myThreads)
            {
                myThread.Join();
            }
            // In ra thông điệp sau khi các mạch trình xong việc
            Console.WriteLine("All my threads are done.");
        }

        public 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);
                }
            }
            finally
            {
                Monitor.Exit(this);
            }
        }
    }
}
```

```

        // ta đã tăng rồi, cho thread khác chơi với Monitor
        Monitor.Pulse(this);
    }
}
finally
{
    Console.WriteLine("[{0}]Existing... ",
                      Thread.CurrentThread.Name);
    Monitor.Exit(this);
}
}

public void Decrementer()
{
    try
    {
        // đồng bộ hóa vùng mã này
        Monitor.Enter(this);

        // nếu counter chưa đến 10 thì giải phóng monitor cho
        // các thread chờ đợi khác, nhưng vào hàng chờ đến phiên
        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);
    }
}

private long counter = 0;
}
}
}
*****

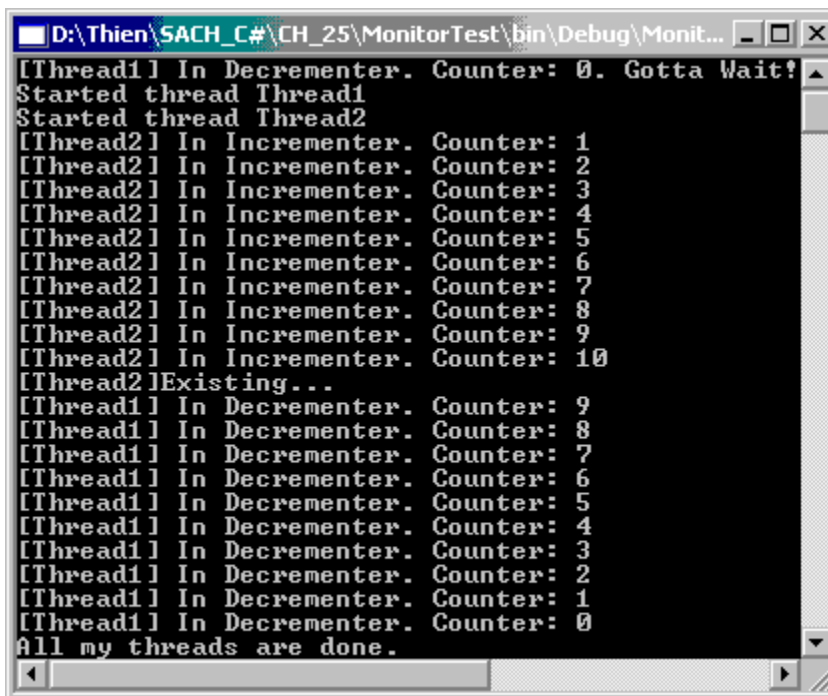
```

Hình 6-06 cho thấy kết xuất của thí dụ 6-04. Trong thí dụ này, **Decrementer** xuất phát trước tiên, nhưng nhận ra rằng nó phải chờ. Rồi bạn thấy là **Thread2** bắt đầu chạy. Chỉ khi **Thread2** pulse thì **Thread1** bắt đầu làm việc.

Bạn thử vài thí nghiệm đối với đoạn mã này. Trước tiên, cho comment out triệu gọi hàm **Pulse()**. Bạn thấy là **Thread1** không bao giờ tiếp tục lại. Không có **Pulse()** thì không có tín hiệu đối với các mạch trình đang chờ.

Thử nghiệm thứ hai bạn cho viết lại Incrementer cho pulse và exit monitor sau mỗi lần tăng:

```
public void Incrementer()
{
    try
    {
        while (counter < 10)
        {
            Monitor.Enter(this);
            long temp = counter;
            temp++;
            Thread.Sleep(1);
            counter = temp;
            Console.WriteLine("[{0}] In Incrementer. Counter: {1}",
                Thread.CurrentThread.Name, counter);
            Monitor.Pulse(this);
            Monitor.Exit(this);
        }
    }
}
```



Hình 6-06: Sử dụng Monitor

Bạn cũng cho viết lại **Decrementer**, thay câu lệnh if bởi câu lệnh while và cho trị từ 10 xuống 5:

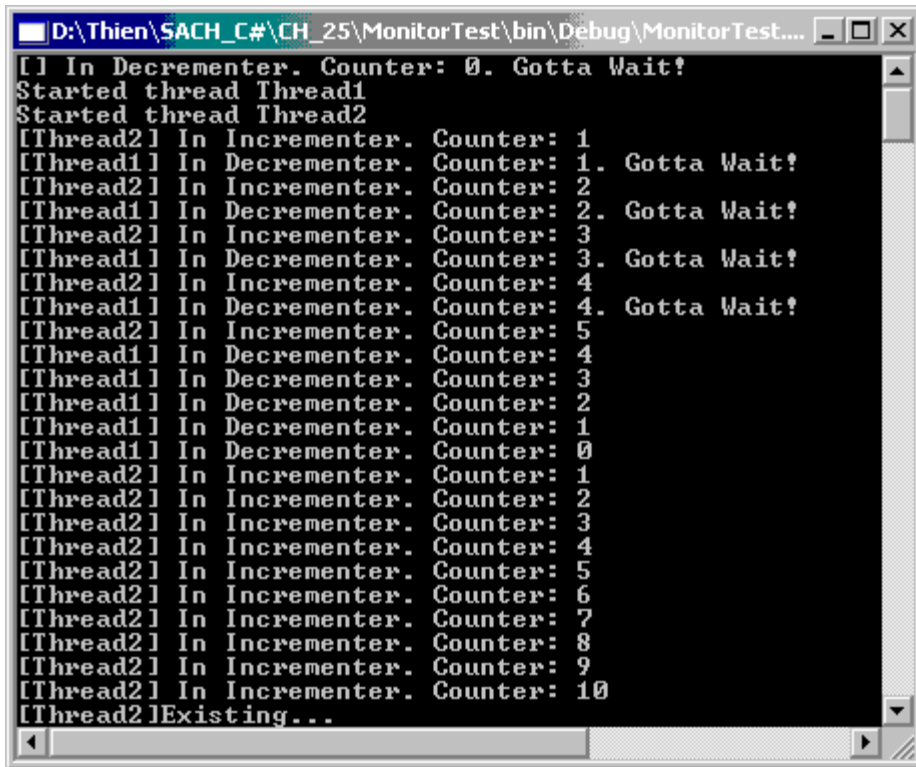
```
// if (counter < 10)
```



```
while (counter < 5)
```

Hình 6-07 cho thấy kết xuất của những thay đổi này:

Tác dụng rõ ràng của những thay đổi này là làm cho **Thread2, Incrementer**, cho **pulse Decrementer** sau mỗi lần tăng. Trong khi trị nhỏ thua 5, **Decrementer** tiếp tục chờ và một khi trị lớn hơn 5 thì **Decrementer** bắt đầu chạy cho đến cuối. Khi nó chạy thì mạch trình **Incrementer** giờ đây có thể chạy. Kết quả là hình 6-07.



```
D:\Thien\SACH_C#\CH_25\MonitorTest\bin\Debug\MonitorTest...
[ ] In Decrementer. Counter: 0. Gotta Wait?
Started thread Thread1
Started thread Thread2
[Thread2] In Incrementer. Counter: 1
[Thread1] In Decrementer. Counter: 1. Gotta Wait?
[Thread2] In Incrementer. Counter: 2
[Thread1] In Decrementer. Counter: 2. Gotta Wait?
[Thread2] In Incrementer. Counter: 3
[Thread1] In Decrementer. Counter: 3. Gotta Wait?
[Thread2] In Incrementer. Counter: 4
[Thread1] In Decrementer. Counter: 4. Gotta Wait?
[Thread2] In Incrementer. 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
[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]Existing...
```

Hình 6-07: Kết xuất khi viết lại Incrmenter và Decrementer

## 6.3 Điều kiện chạy đua và hiện tượng chết chùm (deadlock)

.NET Library cung cấp khá đầy đủ những hỗ trợ về mặt mạch trình nên ít khi bạn phải tự mình sáng chế ra những mạch trình riêng của mình cũng như tự quản lý bằng tay việc đồng bộ hóa.

Việc đồng bộ hóa đôi khi rắc rối đặc biệt đối với những chương trình phức tạp. Nếu bạn quyết định tự mình tạo những mạch trình, bạn phải đối đầu và giải quyết những vấn đề cổ điển trong việc đồng bộ hóa các mạch trình, chẳng hạn các điều kiện chạy đua (race condition) và hiện tượng chết chum (deadlock).

### 6.3.1 Các điều kiện chạy đua

Một *race condition* (điều kiện chạy đua) hiện hữu khi sự thành công chương trình của bạn tùy thuộc vào thứ tự hoàn thành nhiệm vụ không điều khiển được của hai mạch trình.

Thí dụ, giả sử bạn có hai mạch trình - một chịu trách nhiệm mở một tập tin và mạch trình kia chịu trách nhiệm viết một tập tin. Điều quan trọng là bạn phải kiểm tra mạch trình thứ hai, bảo đảm là mạch trình thứ nhất đã mở tập tin. Nếu không, dưới một vài điều kiện mạch trình thứ nhất sẽ mở tập tin và mạch trình thứ hai sẽ chạy tốt, dưới những điều kiện khác không tiên đoán được, mạch trình thứ nhất chưa mở xong tập tin trước khi mạch trình thứ hai cố viết lên tập tin và bạn phải tung ra một biệt lệ (hoặc tồi tệ, chương trình sẽ “ngủ” luôn). Đây là một race condition mà ta khó lòng phát hiện sai lầm.

Bạn không thể để cho hai mạch trình này hoạt động một cách độc lập; bạn phải bảo đảm **Thread1** sẽ phải hoàn tất công việc trước khi **Thread2** bắt đầu. Muốn thế, bạn phải **Join()** **Thread2** trên **Thread1**. Bạn cũng có thể sử dụng **Monitor** và **Wait()** đối với những điều kiện thích ứng trước khi cho **Thread2** tiếp tục.

### 6.3.2 Chết chum (deadlock)

Khi bạn đang chờ một nguồn lực rảnh, bạn có thể gặp phải hiện tượng “chết chum” gọi là deadlock. Trong một deadlock, hai hoặc nhiều mạch trình người này chờ người kia giải phóng nguồn lực và không người nào rảnh cả.

Giả sử bạn có hai mạch trình, **ThreadA** và **ThreadB**. **ThreadA** khoá chặt một đối tượng **Employee** (nhân viên), và cố nhận một cái khoá đối với một hàng dữ liệu trên căn cứ dữ liệu. Nhưng xem ra, thì **ThreadB** đã cho khoá chặt hàng dữ liệu này, do đó **ThreadA** đành phải ngồi chờ.

Rất tiếc là **ThreadB** không thể nhả tu hàng dữ liệu cho tới khi nó khoá chặt đối tượng **Employee**, đã bị khoá bởi mạch trình **Thread6**. Như vậy cả hai mạch trình không thể tiến hành công việc và cũng không thể giải phóng nguồn lực mình đang nắm giữ. Hai mạch trình đang chờ nhau trong một deadly embrace (ôm nhau chết chum).

Như đã được mô tả, ta có thể phát hiện và sửa chữa dễ dàng deadlock. Trong một chương trình có nhiều mạch trình chạy cùng lúc, có thể khó lòng chẩn đoán deadlock.

Cách hướng dẫn là nhận lấy tất cả các lock bạn cần đến hoặc cho giải phóng tất cả các khoá hiện bạn đang nắm giữ. Nghĩa là, khi **ThreadA** nhận ra là nó không thể khoá chặt hàng dữ liệu thì nó phải “nhả” đối tượng **Employee** ra. Cũng tương tự như thế, khi **ThreadB** không thể khoá **Employee** thì nó cũng nhả hàng dữ liệu ra. Hướng dẫn quan trọng thứ hai là chỉ nên khoá chặt một vùng nhỏ đoạn mã và đừng giữ khoá quá lâu.

## Chương 7

# Tương tác với Unmanaged Code<sup>17</sup>

Tới đây, chúng tôi hy vọng là bạn đã nắm vững ngôn ngữ C# và những dịch vụ cốt lõi mà sàn diễn .NET cung cấp cho bạn. Tuy nhiên, trong thực tế trong nhiều thập niên qua, các tổ chức sản xuất cũng như tiêu thụ phần mềm đã mất không biết bao nhiêu công sức và tiền của để phát triển và tích lũy những kỹ thuật như ActiveX control, COM, MFC, ATL, Visual Basic 6.0, và Windows DNA cổ điển, để có thể bỏ ngang một cách dễ dàng quay qua sử dụng ngay liền C# và mô hình đối tượng .NET. Do đó, nếu bạn ở trong một tổ chức sản xuất kinh doanh hoặc hành chính sự nghiệp, những chương trình được viết ra và chạy tốt (mà bây giờ người ta gọi là “unmanaged code”) trước khi C#/.NET xuất hiện, không thể vất vào xọt rác viết lại theo ngôn ngữ C# chạy trên nền .NET, để được tiếng là tiên tiến hiện đại. Nếu .NET được chọn như là sàn diễn cần tiếp tục, thì bạn phải có cách tương tác một cách “nhỏ nhẹ” với các hệ thống phần mềm cũ được gọi là legacy system (hệ thống “gia tài” để lại của người đi trước).

Chương này bắt đầu xét đến việc làm thế nào các kiểu dữ liệu .NET có thể truy cập các hàm Win32 API thô sơ sử dụng đến một dịch vụ được gọi là **PInvoke** (Platform Invoke). Tiếp theo, chúng tôi sẽ đề cập đến đề tài khá lý thú được gọi là “**.NET to COM Interoperability**” và **Runtime Callable Wrapper** (RCW) nghĩa là làm thế nào từ assembly .NET có thể truy cập các cấu kiện COM cổ điển. Phần kế tiếp của chương này sẽ đề cập đến trường hợp ngược lại: một kiểu dữ liệu COM liên lạc với một kiểu dữ liệu .NET sử dụng một CCW (**COM Callable Wrapper**). Cuối cùng, chúng tôi sẽ xét đến tiến trình xây dựng những kiểu dữ liệu managed có thể tương tác với những dịch vụ được cung cấp bởi lớp runtime COM+ (nghĩa là object pooling, object constructor string, v.v..).

## 7.1 Tìm hiểu vấn đề Interoperability<sup>18</sup>

Khi bạn tạo những assembly sử dụng một trình biên dịch “ăn ý” với .NET, thì coi như bạn đã tạo ra một “managed code” có thể chạy bởi CLR (Common Language Runtime). Managed code đem lại cho bạn một số lợi điểm chẳng hạn quản lý ký ức tự

---

<sup>17</sup> Unmanaged code là đoạn mã nằm ngoài vòng cương toả của CLR, CLR không quản lý được.

<sup>18</sup> Interoperability tạm hiểu là “tính khả dĩ hoạt động liên thông giữa hai hệ thống”.

động, hệ thống kiểu dữ liệu thống nhất CTS, những assembly “tự biên tự diễn”, v.v.. Như bạn đã biết, .NET assembly có một cấu trúc nội tại khá đặc biệt. Ngoài các chỉ thị IL, và type metadata, assembly còn chứa một manifest mô tả trọn vẹn một assembly đồng thời có thể sưu liệu bất cứ những assembly nào nằm ngoài được triệu gọi đến trong assembly này.

Nằm ở đầu phía kia là các COM server cổ điển (là những “unmanaged code”) không “dây mơ rễ má” với .NET assembly ngoài việc cùng mang cái đuôi (extension).DLL hoặc .EXE. Trước tiên, COM server chứa mã máy mang tính đặc thù sàn diễn, chứ không phải chỉ thị IL. COM server làm việc với một bộ kiểu dữ liệu duy nhất (BSTR, VARIANT, v.v..) được ánh xạ rất khác nhau giữa các ngôn ngữ “ăn ý” với COM. Ngoài ra, ngoài những gì cần thiết đối với COM binary (nghĩa là class factory, registry entry và IDL code) các kiểu dữ liệu COM còn đòi hỏi phải có cái đếm qui chiếu (reference counting) dùng quản lý ký ức tránh việc rò rỉ ký ức.

Do đó, hai kiểu dữ liệu COM và .NET rất ít có cái gì chung chung, và bạn có thể ngạc nhiên là cả hai kiến trúc trên có thể “sống chung hoà bình” tồn tại song song. Trừ khi bạn may mắn làm việc cho một công ty mới được tin học hóa lần đầu tiên sử dụng 100% .NET, còn không chắc chắn là bạn phải xây dựng những giải pháp .NET sử dụng đến những gia tài để lại kiểu COM. Có thể bạn sẽ phải xây dựng một hoặc hai COM server cần liên lạc với một .NET assembly vừa mới ra lò. Nghĩa là, trong tương lai và trong một thời gian dài, COM và .NET phải học sống chung với nhau<sup>19</sup>. Chương này sẽ xem xét đến những vấn đề được đặt ra do việc managed code .NET và unmanaged code COM sống chung hoà hợp với nhau. Nhìn chung, .NET Framework hỗ trợ các kiểu interoperability như sau:

- Các kiểu dữ liệu .NET triệu gọi C DLL (nghĩa là Win32 API hoặc custom DLL) sử dụng *.NET platform invoke facility* (gọi tắt PInvoke). Xem mục 7.3.
- Các kiểu dữ liệu .NET triệu gọi các kiểu dữ liệu COM.
- Các kiểu dữ liệu COM triệu gọi các kiểu dữ liệu .NET.
- Các kiểu dữ liệu .NET sử dụng các dịch vụ COM+.

Như bạn sẽ thấy suốt chương này .NET SDK sẽ cung cấp cho bạn một số công cụ giúp bạn bắt nhịp cầu giữa những kiến trúc duy nhất này. Ngoài ra, các lớp cơ bản thư viện .NET còn định nghĩa một số kiểu dữ liệu chỉ dành riêng cho mục đích liên thông ngôn ngữ (interoperability) này mà thôi.

---

<sup>19</sup> Giống như Israel và Palestine

## 7.2 Tìm hiểu về Namespace **System.Runtime.InteropServices**

Khi bạn sử dụng đến các dịch vụ .NET interoperability, thì gián tiếp hoặc trực tiếp bạn tương tác với các kiểu dữ liệu được định nghĩa trong namespace **System.Runtime.InteropServices**. Bảng 7-01 cho liệt kê một số lớp cốt lõi của namespace này.

**Bảng 7-01: Các lớp cốt lõi của *System.Runtime.InteropServices***

Các lớp thành viên	Mô tả
<b>ClassInterfaceAttribute</b>	Lớp này dùng kiểm soát cách thế nào một kiểu dữ liệu managed trung ra những thành viên public của nó đối với COM client.
<b>ComRegisterFunctionAttribute</b> <b>ComUnregisterFunctionAttribute</b>	Lớp này có thể được gắn liền với những hàm hành sự custom cho biết chúng phải được triệu gọi khi assembly được đăng ký sử dụng (hoặc đăng ký bị gỡ bỏ) để dùng với COM.
<b>ComSourceInterfacesAttribute</b>	Lớp này nhận diện danh sách các giao diện là nguồn các tình huống đối với lớp (nghĩa là outbound interface).
<b>DispIdAttribute</b>	Đây là custom attribute cho biết mã nhận diện COM DISPID của một hàm hành sự, vùng mục tin hoặc thuộc tính.
<b>DllImportAttribute</b>	Lớp này được sử dụng bởi các dịch vụ <b>PInvoke</b> để triệu gọi unmanaged code.
<b>GuidAttribute</b>	Lớp này được dùng để định nghĩa một mã nhận diện GUID cụ thể đối với một lớp, giao diện hoặc type library.
<b>IDispatchImplAttribute</b>	Lớp này cho biết thi công <b>IDispatch</b> nào mà CLR sẽ dùng đến khi trung ra dual interface và dispinterface.
<b>InterfaceTypeAttribute</b>	Lớp này điều khiển việc một managed interface sẽ được trung ra thế nào trước COM client (được dẫn xuất từ <b>IDispatch</b> hoặc từ <b>IUnknown</b> ).
<b>OutAttribute</b> <b>InAttribute</b>	Lớp này được dùng trên một thông số hoặc trên một vùng mục tin cho biết dữ liệu phải được marshalling ra từ phía bị gọi (callee) ngược về phía triệu gọi hoặc ngược lại.
<b>ProgIdAttribute</b>	Đây là custom attribute cho phép người sử dụng khai báo mã nhận diện chương trình (prog ID) của một kiểu dữ liệu .NET.

Như bạn có thể thấy, phần lớn các lớp này là những attribute dùng marshalling dữ liệu giữa kiểu dữ liệu .NET và kiểu dữ liệu COM. Lẽ dĩ nhiên namespace này cũng định nghĩa một số giao diện, enumeration và structure. Muốn sử dụng .NET interoperability, đầu dự án bạn phải ghi chỉ thị using như sau:

```
using System.Runtime.InteropServices;
```

Lộ trình của ta bây giờ là trước tiên xem xét việc sử dụng **PInvoke**.

## 7.3 Tương tác với .DLL viết theo C#

Các dịch vụ **PInvoke** (Platform Invocation<sup>20</sup>) đem lại cho managed code một thể thức triệu gọi các hàm unmanaged được thi công trên một DLL viết theo ngôn ngữ C cổ điển (non-COM). Bằng cách sử dụng **PInvoke**, lập trình viên .NET được “bao tiêu” khỏi công việc dò tìm và triệu gọi đúng hàm export. Ngoài ra, **PInvoke** sẽ lo việc marshalling dữ liệu managed (nghĩa là integer, string, array và structure) đi đi về về với phía “đối tác” unmanaged code.

Một sử dụng điển hình **PInvoke** là cho phép các cấu kiện .NET tương tác với các hàm Win32 API ở dạng thô sơ. Như bạn đã biết, thư viện lớp cơ bản .NET được tạo ra là để khởi vọc vào các chức năng cốt lõi của Windows API cấp thấp. Do đó, **PInvoke** cho phép bạn vọc Win32 API ở dạng thô sơ, bao gồm **user32.dll**, **gdi32.dll** và **kernel32.dll**. **PInvoke** cũng cho phép bạn truy xuất các hàm export được định nghĩa trên custom DLL. Như vậy, nếu bạn thừa hưởng một đoạn mã C nằm ở dạng .DLL, thì bạn sẽ vui mừng khi biết các cấu kiện .NET của bạn có thể sử dụng các DLL C này.

Nói tóm lại, **PInvoke** lúc ban đầu cho ra để có thể truy cập Windows API, nhưng bạn có thể dùng nó để trung ra các hàm trên bất cứ DLL nào.

Để minh họa việc sử dụng **PInvoke**, bạn có thể xây dựng một lớp C# cho triệu gọi hàm **MessageBox() API**. Thí dụ 7-01 sau đây là đoạn mã:

### Thí dụ 7-01: Sử dụng PInvoke

```
*****
using System;
using System.Runtime.InteropServices;    // phải có để sử dụng PInvoke

namespace PInvokeExample
{
```

<sup>20</sup> Platform invocation có nghĩa là triệu gọi sàn diễn

```

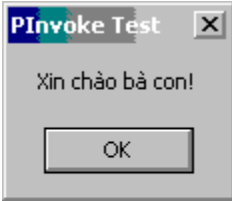
public class PInvokeClient
{
    // Hàm Win32 MessageBox() nằm ở user32.dll
    [DllImport("user32")]
    public static extern int MessageBox(int hWnd, String pText,
                                       String pCaption, int uType);

    public static int Main(string[] args)
    {
        String pText = "Xin chào bà con!";
        String pCaption = "PInvoke Test";
        MessageBox(0, pText, pCaption, 0);

        return 0;
    }
}

```

\*\*\*\*\*



**Hình 7-01:**  
**Message Box**

Hình 7-01 là kết xuất message box

Tiến trình triệu gọi một DLL kiểu C bắt đầu bằng cách khai báo hàm bạn muốn triệu gọi sử dụng các từ chốt **static extern**. Bước này bắt buộc phải có. Bạn để ý là khi bạn khai báo hàm prototype C, bạn phải liệt kê kiểu dữ liệu trả về của hàm, và các đối mục theo kiểu dữ liệu managed. Do đó, bạn không được viết các bản dây char\* hoặc wchar\_t\* mà là kiểu dữ liệu **System.String**.

Một khi bạn đã cho vào khuôn mẫu (prototyped) hàm hành sự bạn muốn triệu gọi, bước kế tiếp là khai báo hàm thành viên này với attribute [DllImport...]. Tối thiểu bạn phải khai báo tên của API32 DLL chứa đựng hàm mà bạn muốn triệu gọi, giống như sau:

```

[DllImport("User32.dll")]
public static extern int MessageBox(int hWnd, String pText,
                                   String pCaption, int uType);

```

Lớp **DllImportAttribute** định nghĩa một tập hợp những vùng mục tin public mà ta có thể khai báo để có thể cấu hình tiến trình gắn kết đối với hàm export. Bảng 7-02 mô tả các vùng mục tin của lớp **DllImportAttribute**.

**Bảng 7-02: Các vùng mục tin của lớp DllImportAttribute.**

Các vùng mục tin	Mô tả
<b>CallingConvention</b>	Dùng thiết lập qui ước triệu gọi được dùng để trao qua các đối mục hàm hành sự.
<b>CharSet</b>	Cho biết các đối mục kiểu chuỗi đối với hàm hành sự phải được marshalling thế nào.
<b>EntryPoint</b>	Cho biết tên hoặc số thứ tự (ordinal) của hàm hành sự phải được marshalling.



<b>ExactSpelling</b>	Như bạn sẽ thấy, <b>PInvoke</b> cố gắng so khớp tên hàm bạn khai báo với tên thực sự được prototyped. Nếu vùng mục tin này được cho về <b>true</b> , bạn cho biết là tên của entry point trên unmanaged.dll phải khớp đúng với tên bạn trao qua.
<b>PreserveSig</b>	Khi cho về <b>true</b> (là trị mặc nhiên) dấu ấn của hàm hành sự unmanaged <i>không được</i> biến đổi thành dấu ấn managed sẽ trả về một HRESULT và một đối mục bổ sung [out, retval] đối với trị trả về.
<b>SetLastError</b>	Nếu vùng mục tin này được cho về <b>true</b> có nghĩa là phía triệu gọi có thể gọi hàm Win32 <b>GetLastError()</b> để xác định liệu xem một sai lầm có xảy ra hay không trong khi thi hành hàm hành sự. Trị mặc nhiên là <b>false</b> .

Muốn đặt để các trị này đối với thể hiện đối tượng **DllImportAttribute** hiện hành, bạn chỉ cần kê khai mỗi cặp name/value vào trong hàm constructor của lớp. Nếu bạn xem lại định nghĩa của lớp **DllImportAttribute**, bạn sẽ thấy hàm constructor của lớp này chỉ nhận một thông số duy nhất kiểu dữ liệu **System.String** như sau:

```
class DllImportAttribute
{ // Hàm constructor lấy một chuỗi chứa tất cả các trị vùng mục tin
  public DllImportAttribute(string val);
  . . .
}
```

Bạn có thể liệt kê các cặp name/value không theo thứ tự nào cả. Lớp **DllImportAttribute** sẽ phân tích ngữ nghĩa chuỗi trên và dùng các trị để đặt để trạng thái dữ liệu nội tại.

### 7.3.1 Khai báo vùng mục tin ExactSpelling

Vùng mục tin thứ nhất cần xem xét là **ExactSpelling**, dùng kiểm tra liệu xem hàm managed có giống đúc tên của hàm unmanaged hay không. Thí dụ, có thể bạn đã biết không có hàm nào mang tên **MessageBox** trên Win32 API. Thay vào đó, ta có phiên bản ANSI (MessageBoxA) và phiên bản Unicode (MessageBoxW). Nếu bạn khai báo một hàm hành sự mang tên MessageBox, bạn có thể giả sử đúng là trị mặc nhiên của **ExactSpelling** là false. Tuy nhiên, nếu bạn cho trị này về **true**, bạn sẽ có dòng lệnh sau đây:

```
[DllImport("user32", ExactSpelling = true)]
public static extern int MessageBox(...); // không xong
```

Bây giờ bạn sẽ nhận một biệt lệ **EntryPointNotFoundException** cho biết là không có hàm nào mang tên **MessageBox** trên User32.dll cả. Như bạn có thể thấy, **ExactSpelling** thực chất cho phép bạn hơi “luời” một chút, và phớt lờ chi tiết cái đuôi A hoặc W trên MessageBox. Tuy nhiên, rõ ràng cuối cùng là **PInvoke** không cần phải giải

quyết tên chính xác của hàm mà bạn triệu gọi. Khi bạn để **ExactSpelling** ở yên tại trị mặc nhiên (false) thì cái đuôi A sẽ gắn vào tên hàm hành sự dưới môi trường Ansi, hoặc W dưới môi trường Unicode. Nói cách khác, bạn có thể khỏi kê khai ra **ExactSpelling**.

## 7.3.2 Khai báo vùng mục tin CharSet

Muốn khai báo rõ ra bộ ký tự (character set) được sử dụng để marshalling dữ liệu giữa managed code và DLL export thô, bạn có thể cho đặt đề trị của vùng mục tin **CharSet** sử dụng một trong những trị enumeration **CharSet** như theo bảng 7-03

**Bảng 7-03: Các trị enum CharSet**

Các trị enum	Mô tả
<b>Ansi</b>	Cho biết các chuỗi sẽ được marshalling như là những ký tự ANSI 1 byte.
<b>Auto</b>	Báo cho <b>PInvoke</b> marshalling đúng đắn chuỗi như theo yêu cầu của sàn diễn đích (Unicode trên WINNT/Win2000 và ANSI trên Win9x).
<b>None</b>	(Trị mặc nhiên) Cho biết bạn không khai báo phải marshalling các chuỗi thế nào và yêu cầu CLR tự giải quyết lấy.
<b>Unicode</b>	Cho biết các chuỗi sẽ được marshalling như là những ký tự Unicode 2 byte.

Để lấy thí dụ, muốn tất cả các chuỗi phải được marshalling theo Unicode (biết trước là không chạy trên sàn diễn Win95, Win98 hoặc WinME), bạn có thể viết như sau:

```
[DllImport("user32", ExactSpelling = true, CharSet=CharSet.Unicode)]
public static extern int MessageBoxW(...);
```

Nhìn chung, sẽ an toàn hơn khi ta cho trị của CharSet về CharSet.Auto (hoặc đơn giản chấp nhận CharSet.None, trị mặc nhiên). Theo cách này, các thông số kiểu chữ sẽ được marshalling một cách đúng đắn không cần biết đến sàn diễn.

## 7.3.3 Khai báo vùng mục tin CallingConvention và EntryPoint

Hai vùng mục tin còn lại đáng ta quan tâm là **CallingConvention** và **EntryPoint**. Như có thể bạn đã biết, các hàm Win32 API có vô số định nghĩa kiểu dữ liệu (typedef) cho biết các thông số phải được trao qua hàm thế nào (C declaration, fast call, standard call, v.v.). Bạn có thể cho đặt đề vùng mục tin **CallingConvention** sử dụng bất cứ trị nào của enumeration **CallingConvention** (Cdecl, FastCall, StdCall, ThisCall, WinApi), với trị mặc nhiên là StdCall, nên bạn có thể phớt lờ việc đặt đề tường minh vùng mục tin này (vì đây là calling convention Win32 thông dụng nhất).

Cuối cùng là việc đặt để vùng mục tin **EntryPoint**, cho biết điểm đột nhập Theo mặc nhiên, trị này là giống như tên của hàm mà bạn đang prototyping. Như vậy, khai báo sau đây cho biết điểm đột nhập được hiểu ngầm là `MessageBoxW`:

```
[DllImport("user32", ExactSpelling = true, CharSet=CharSet.Unicode)]
public static extern int MessageBoxW(...);
```

Muốn thiết lập một alias (bí danh) đối với hàm được xuất khẩu, bạn có thể khai báo tên thật sử dụng vùng mục tin **EntryPoint**, thật sự đổi tên hàm để dùng trong managed code. Rõ ràng đây là cách tốt nhất để tránh việc đụng độ tên có thể xảy ra. Để minh họa, sau đây là phiên bản cuối cùng của thí dụ về **PInvoke**, thí dụ 7-02, cho ánh xạ hàm `MessageBoxW()` về một hàm bí danh thân thiện hơn mang tên `DisplayMessage`:

### ***Thí dụ 7-02: Sử dụng PInvoke, phiên bản cuối cùng***

```
*****
using System;
using System.Runtime.InteropServices;

namespace PInvokeExample
{
    public class PInvokeClient
    {
        // Ánh xạ hàm Win32 MessageBoxW() về DisplayMessage
        [DllImport("user32", ExactSpelling = true,
        CharSet = CharSet.Unicode, EntryPoint = "MessageBoxW")]
        public static extern int DisplayMessage(int hWnd, String pText,
        String pCaption, int uType);

        public static int Main(string[] args)
        {
            String pText = "Xin chào bà con!";
            String pCaption = "PInvoke Test";
            DisplayMessage(0, pText, pCaption, 0);
            return 0;
        }
    }
}
*****
```

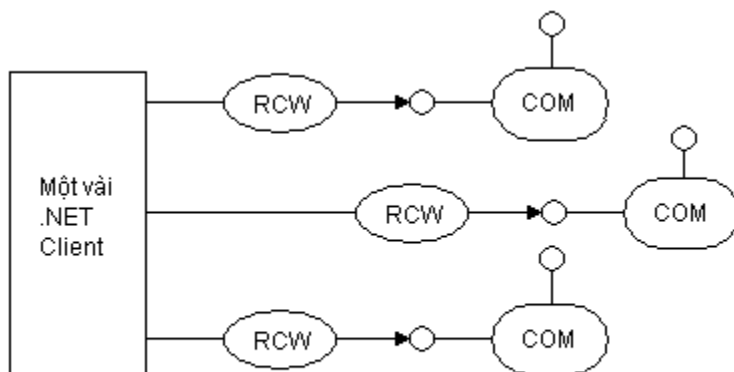
Để kết luận, khi bạn sử dụng **PInvoke**, coi như bạn trói tay .NET với hệ điều hành Windows. Do đó, khi đem chương trình .NET của bạn qua dùng trên sàn diễn Unix chẳng hạn, thì chương trình .NET có sử dụng **PInvoke** sẽ mất đi tính “du di” (portability). Cuối cùng, khi muốn sử dụng **PInvoke**, bạn nên kiểm tra lại xem chức năng Win32 API bạn muốn trưng ra, hiện có đâu đó trên các lớp cơ bản .NET hay không. Thông thường các chức năng thông dụng và hữu ích nhất của Win32 API đều có trên .NET.

## 7.4 Tìm hiểu .NET to COM Interoperability

Bước kế tiếp mà ta phải tìm hiểu là: Làm thế nào managed code C# có thể sử dụng kiểu dữ liệu unmanaged COM. Khi bạn bắt đầu xây dựng những giải pháp .NET, có thể những assembly mới của bạn muốn sử dụng những COM server hiện hữu. Muốn thế, phải có một tầng lớp trung gian cho trưng ra những kiểu dữ liệu COM như là tương đương với .NET. Tiến trình ánh xạ phải thông suốt, như vậy mới có thể cho phép các kiểu dữ liệu .NET đối xử các kiểu dữ liệu COM như là của riêng mình.

### *Runtime Callable Wrapper (RCW)*

Cái “hộp đen” mà chúng tôi muốn nói đến mang tên RCW (Runtime Callable Wrapper<sup>21</sup>). RCW có thể được xem như là một proxy đối với các lớp thật sự COM (thường được gọi là coclass, tắt chữ Com Class). Nói cách khác, *mỗi coclass được truy cập bởi một .NET client đòi hỏi phải có một RCW tương ứng*. Như vậy, nếu bạn có một ứng dụng .NET đơn độc sử dụng 3 COM coclass, bạn sẽ đi đến 3 RCW riêng biệt ánh xạ các triệu gọi .NET về những yêu cầu COM. Hình 7-02 minh họa khái niệm RCW:



**Hình 7-02: Hàm RCW hoạt động như là proxy đối với các COM coclass.**

Bạn nên để ý là *có một RCW đơn độc cho mỗi đối tượng COM*, không cần biết đến bao nhiêu giao diện riêng biệt mà .NET client có thể nhận được từ một lớp COM nào đó. Sử dụng kỹ thuật này, RCW có thể duy trì sự nhận diện đúng đắn COM (và reference count) của kiểu dữ liệu COM.

Điểm tốt lành là RCW sẽ được tự động kết sinh sử dụng đến một công cụ mang tên **tlbimp.exe** (type library importer). Thật ra, khi bạn dùng khung đối thoại **Add Reference** trên IDE để đưa một qui chiếu về một COM DLL, thì Visual Studio .NET IDE ở sau hậu trường đã kết sinh một .NET proxy component RCW đối với COM DLL và đưa một bản sao của COM DLL vào thư mục của dự án .NET. Nhờ RCW, .NET client không biết là mình đang triệu gọi COM DLL mà chỉ biết là nó đang nói chuyện với proxy và nhận dữ liệu mà proxy chuyển lại từ COM DLL.

<sup>21</sup> Dịch trắng trợn là “vỏ bọc có thể triệu gọi được vào lúc chạy”. Bạn có hiểu gì không ?

Một điểm tốt lành khác là các COM coclass được kế thừa không cần phải thay đổi gì bởi một ngôn ngữ ẩn ý với .NET.

## 7.4.1 Cho trung COM Type như là .NET tương đương

RCW có nhiệm vụ trung các kiểu dữ liệu COM thành những .NET tương đương. Lấy một thí dụ đơn giản, giả sử bạn có một hàm trong một giao diện COM được định nghĩa trên IDL như sau:

```
// định nghĩa IDL đối với một hàm COM
HRESULT DisplayThisString([in] BSTR msg);
```

RCW sẽ cho trung hàm hành sự này đối với một .NET client như sau, sử dụng các kiểu dữ liệu bản sinh C#:

```
// ánh xạ C# đối với một COM IDL method
void DisplayThisString(String msg);
```

Phần lớn các kiểu dữ liệu COM (kể cả tất cả các kiểu dữ liệu tương thích [oleautomation]) đều có một tương đương .NET tương ứng. Bảng 7-04 sau đây cho thấy ánh xạ giữa kiểu dữ liệu COM IDL và kiểu dữ liệu .NET tương đương (kể cả C# alias tương ứng).

**Bảng 7-04: Bảng ánh xạ kiểu dữ liệu bản sinh COM tương đương với kiểu dữ liệu .NET**

COM(IDL) data type	.NET data type	C# Alias
char, boolean, small	System.Sbyte	sbyte
wchar_t, short	System.Int16	short
long, int	System.Int32	int
hyper	System.Int64	long
unsigned char, byte	System.Byte	byte
unsigned short	System.UInt16	ushort
unsigned long, unsigned int	System.UInt32	uint
unsigned hyper	System.UInt64	ulong
single	System.Single	float
double	System.Double	double
VARIANT_BOOL	n/a	bool
HRESULT	System.Int32	int
BSTR	System.String	string
LPSTR or char*	System.String	string

LPWSTR hoặc wchar_t	System.String	string
VARIANT	System.Object	object
DECIMAL	System.Decimal	n/a
DATE	System.DateTime	n/a
GUID	System.Guid	n/a
CURRENCY	System.Decimal	n/a
IUnknown*	System.Object	object
IDispatch*	System.Object	object

Ghi chú: n/a: not announced, không thông báo.

Ngoài ra, bạn cũng nên để ý là nếu bạn có một định nghĩa IDL pointer (nghĩa là int\* thay vì int) nó sẽ ánh xạ lên cùng lớp cơ bản và C# alias (nghĩa là System.Int32/int). Về sau trong chương này, khi bạn xây dựng ATL COM server, bạn sẽ có cơ hội thấy việc chuyển đổi những kiểu dữ liệu COM khá lý thú chẳng hạn SAFEARRAY và COM enum.

## 7.4.2 Quản lý cái đếm qui chiếu của một CoClass

Một nhiệm vụ quan trọng của RCW là quản lý cái đếm qui chiếu (reference count) của coclass nằm đằng sau. Cơ chế của một reference count là giải quyết việc sử dụng thích ứng các triệu gọi hàm **AddRef()** và **Release()**. Các coclass COM sẽ tự hủy khi phát hiện cái đếm qui chiếu bằng zero, nghĩa là không còn qui chiếu về lớp.

Tuy nhiên, các kiểu dữ liệu .NET không sử dụng cơ chế reference count và do đó một .NET client không bị ép buộc phải triệu gọi hàm **Release()** trên kiểu dữ liệu COM mà nó sử dụng. Để cho mọi phía đều vui vẻ, RCW sẽ che dấu tất cả các qui chiếu giao diện trong nội bộ và sẽ kích hoạt việc giải phóng lớp khi kiểu dữ liệu không được dùng nữa bởi .NET client. Nói tóm lại, .NET client sẽ không bao giờ triệu gọi rõ ra các hàm **AddRef()**, **Release()** và **QueryInterface()** của COM.

## 7.4.3 Che dấu Low-Level COM Interfaces

Vai trò cuối cùng của RCW là tiêu thụ một số giao diện được chọn lọc. Vì RCW làm đủ mọi thứ ở hậu trường nên .NET client có cảm tưởng là mình đang sử dụng một kiểu dữ liệu .NET, do đó RCW phải che dấu những giao diện cấp thấp khác nhau khỏi cho .NET client biết đến. Trong chừng mực nào đó cách tiếp cận của RCW hầu như giống với Visual Basic 6.0.

Thí dụ, khi bạn xây dựng một lớp COM chịu hỗ trợ **IConnectionPointContainer** (và duy trì một hoặc hai subobject chịu hỗ trợ **IConnectionPoint**), coclass này có khả năng phát pháo các tình huống trả về cho COM client. Với C++ COM client, bạn phải tiến hành một số bước để thiết lập một kết nối với đối tượng (nghĩa là xây dựng một sink thì

công giao diện [source], nhận một qui chiếu **IConnectionPoint**, triệu gọi hàm **Advise()**, v.v..).

Visual Basic 6.0 che dấu toàn bộ tiến trình không cho thấy bằng cách sử dụng từ chốt **WithEvents**. Cũng tương tự như thế, RCW che dấu tiến trình chuyển đổi COM khỏi .NET client. Vì RCW che dấu những giao diện cấp thấp, nên bên ngoài, .NET client chỉ thấy tập hợp những giao diện custom thiết đặt bởi coclass. Bảng 7-05 liệt kê cho thấy những giao diện bị che dấu:

**Bảng 7-05: Các giao diện COM bị che dấu**

Giao diện COM bị che dấu	Mô tả
<b>IClassFactory</b>	Cung cấp một thể thức hiện dịch (activate) một lớp COM độc lập khỏi ngôn ngữ và vị trí.
<b>IConnectionPointContainer</b> <b>IConnectionPoint</b>	Cho coclass khả năng gởi đi những tình huống trả về cho một khách hàng đáng quan tâm.
<b>IDispatch</b> <b>IDispatchEx</b> <b>IProvideClassInfo</b>	Dùng trong việc gắn kết trễ (late binding) đối với một coclass.
<b>IEnumVariant</b>	Các lớp COM có thể trưng những collection các kiểu dữ liệu nội tại “bẩm sinh”. Giao diện này cho phép thực hiện việc này.
<b>IErrorInfo</b> <b>ISupportErrorInfo</b> <b>ICreateErrorInfo</b>	Các giao diện này cho COM client và coclass khả năng gởi đi và tiếp nhận những đối tượng sai lầm Error.
<b>IUnknown</b>	Giao diện này quản lý reference count và cho phép client nhận một giao diện riêng rẽ từ coclass.

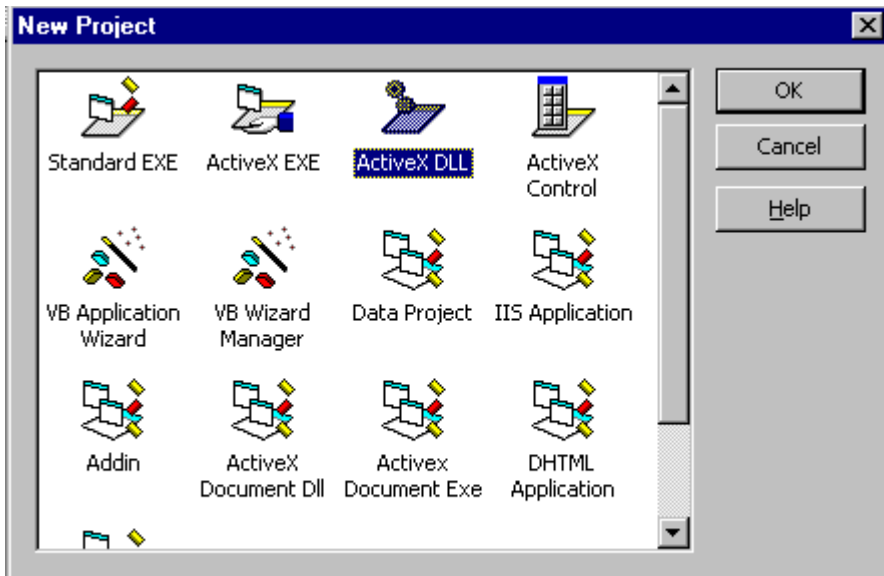
Tới đây coi như bạn đã biết vai trò của RCW. Bây giờ, chúng tôi sẽ xét đến những chi tiết lập trình giữa .NET với COM với một thí dụ đơn giản sử dụng Visual Basic 6.0 (để tạo COM server) và C# (để tạo .NET client). Về sau trong chương này, chúng tôi sẽ triển khai một COM server phức tạp tinh vi hơn sử dụng ATL 3.0.

## 7.4.4 Tạo một COM server rất đơn giản viết theo Visual Basic

Để bạn nắm vững vấn đề .NET to COM Interoperability sử dụng Visual Basic 6.0, ta thử xây dựng một COM server thích ứng cho mang tên **VBCOMServerSimple** chẳng hạn. Bạn bắt đầu cho mở Visual Basic 6.0 và chọn một dự án **ActiveX DLL** từ khung đối thoại **New Project** (một in-process COM server). Xem hình 7-03:

Bạn dùng cửa sổ **Properties**, cho đổi tên dự án thành **VBCOMServerSimple** (hoặc một tên gì đó ngắn hơn) rồi đổi tiếp lớp của bạn thành **CoCalc** (thay vì mặc nhiên là

Class1). Thông tin này sẽ được dùng để xây dựng mã nhận diện chương trình ProgID sử dụng ký hiệu chuẩn *ServerName.ObjectName*.



Hình 7-03: Xây dựng một VB COM Server

Tiếp theo bạn cho mở code window đối với lớp Visual Basic 6.0 **CoCalc** rồi bạn kho vào một hàm toán cộng hơi vô duyên như sau:

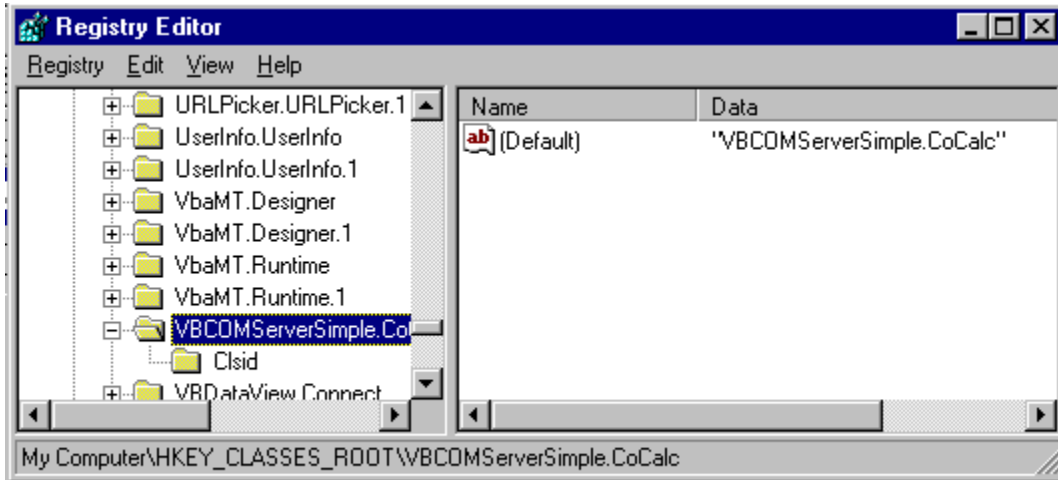
```
` Nhớ cho!!! Đây thật sự một hàm hành sự
` của giao diện mặc nhiên:_CoCalc
Public Function Add(ByVal x As Integer, ByVal y As Integer)As Integer
    Add = x + y
End Function
```

Cuối cùng, bạn cho cất trữ workspace của bạn và cho biên dịch COM server bằng cách ra lệnh **File | Make VBCOMServerSimple**. Mục chọn này sẽ tự động đăng ký vào system registry. Hình 7-04 cho thấy ProgID kết quả. Bạn sử dụng trình tiện ích regedit.exe trên command line để có hình 7-04 kể trên.

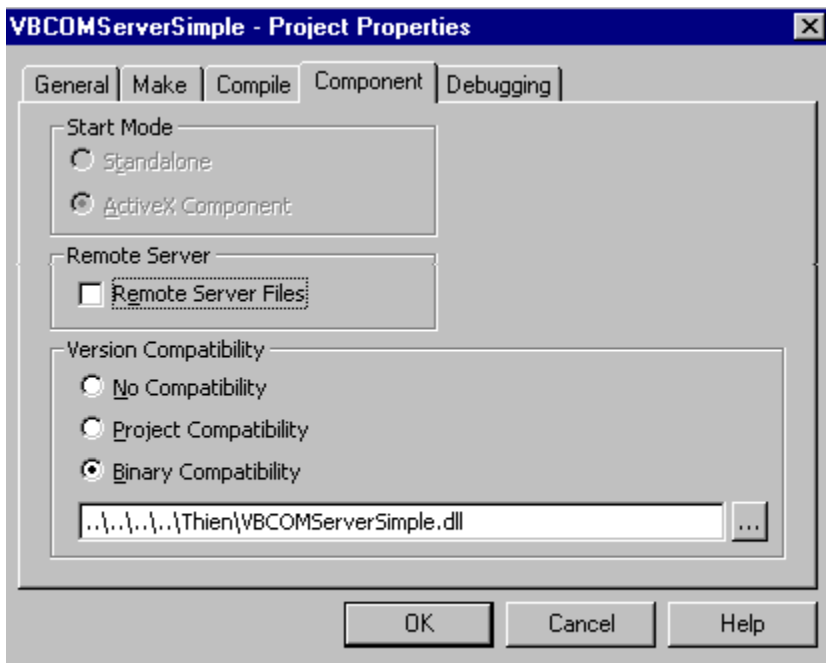
Kết quả cuối cùng là một COM server mới chứa một coclass (CoCalc) duy nhất thi công một giao diện đơn độc [default] mang tên **\_CoCalc**.

Trước khi bạn đóng lại dự án Visual Basic, bạn ra lệnh **Project | Properties** để cho hiện lên khung đối thoại **Project Properties** (hình 7-05) và cho nút radio **Binary Compatibility** về ON đối với COM server này. Việc này báo cho Visual Basic biết là ngưng kết sinh mã GUID mới sau mỗi lần biên dịch.





Hình 7-04: ProgID của COM type



Hình 7-05: Ngừng kết sinh VB GUID

Bạn cho đăng ký **VBCOMServerSimple.dll** này sử dụng trình tiện ích **Regsvr32.exe** trên command line như sau:

```
regsvr32 d:\thien\sach_C#\ch_26\VBCOMServerSimple.dll
```

và một message box sẽ hiện lên cho biết việc đăng ký thành công hay không. Bạn kiểm tra lại bằng cách ra lệnh trên Visual Studio .NET **Project | Add Reference | Tab COM** xem **VBCOMServerSimple** đã vào chưa. Nếu nó hiện diện thì coi như xong.

#### 7.4.4.1 *Quan sát IDL được kết sinh đối với Visual Basic COM Server*

Trên IDE, bạn cho mở **OLE/COM Object Viewer** từ trình đơn **Tools**. Khi khung đối thoại hiện lên bạn ra lệnh **File | ViewTypeLib..** rồi chọn tập tin **VBCOMServerSimple**, thì khung đối thoại ITypeLib Viewer hiện lên (hình 7-06):

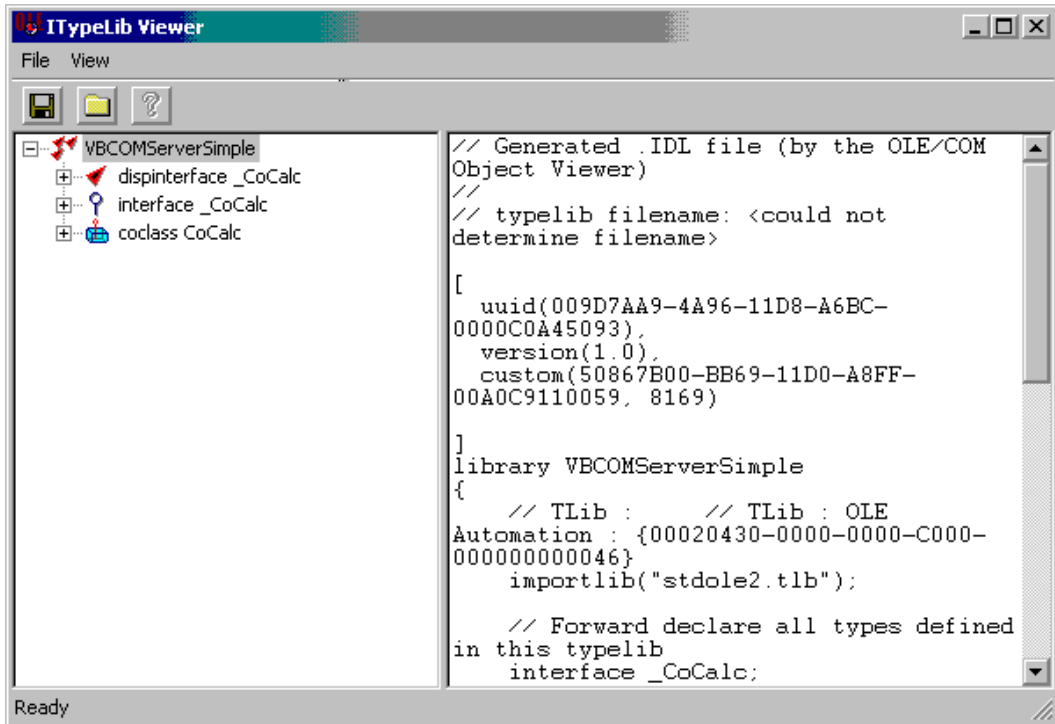
Sau đây là nội dung của tập tin IDL được kết sinh:

```
// Generated .IDL file (by the OLE/COM Object Viewer)
//
// typelib filename: <could not determine filename>

[
    uuid(009D7AA9-4A96-11D8-A6BC-0000C0A45093),
    version(1.0),
    custom(50867B00-BB69-11D0-A8FF-00A0C9110059, 8169)
]
library VBCOMServerSimple
{
    // TLib:    // TLib: OLE Automation: {
                                     00020430-0000-0000-C000-0000000000046}
    importlib("stdole2.tlb");

    // Forward declare all types defined in this typelib
    interface _CoCalc;
    [
        odl, uuid(009D7AA7-4A96-11D8-A6BC-0000C0A45093),
        version(1.0), hidden, dual, nonextensible, oleautomation
    ]
    interface _CoCalc: IDispatch
    {
        [id(0x60030000)]
        HRESULT Add([in] short x, [in] short y, [out, retval] short* );
    };

    [
        uuid(009D7AAB-4A96-11D8-A6BC-0000C0A45093),
        version(1.0)
    ]
    coclass CoCalc {
        [default] interface _CoCalc;
    };
};
```



Hình 7-06: Tập tin IDL được kết sinh (hiển thị bởi OLE/COM Object Viewer)

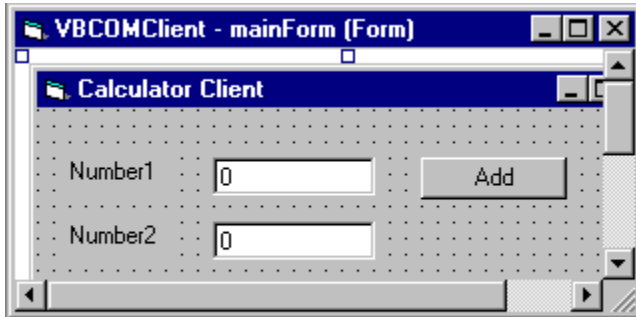
Như bạn có thể thấy, Visual Basic bao giờ cũng cấu hình hóa các giao diện custom của bạn như là [dual] interface (giao diện hai mặt). Việc này cho coclass có khả năng bị thao tác bởi những ngôn ngữ kịch bản khác nhau thông qua giao diện IDispatch. Với việc này, coi như Visual Basic COM server hoàn tất. Bây giờ ta thử trải nghiệm nó với một Visual Basic COM client đơn giản.

## 7.4.5 Xây dựng một Visual Basic COM Client đơn giản

Bạn cho mở Visual Basic lại một lần nữa, nhưng lần này bạn chọn dự án Standard EXE, cho đặt tên là **VBCOMClient**. Tiếp theo, bạn thêm một qui chiếu về **VBCOMServerSimple**, bằng cách ra lệnh **Project | References...** rồi chọn check ô mang tên **VBCOMServerSimple**.

Bạn tạo một giao diện GUI đơn giản gồm hai text box (txtNumb1 và txtNumb2) với label và một nút <Add> như theo hình 7-07. Đoạn mã duy nhất mà bạn cần viết là thủ lý tình huống nút <Add> bị ấn xuống. Bạn tạo một thể hiện của lớp CoCalc rồi chuyển cho hàm hành sự Add trị của hai ô text box. Để đơn giản, ta cho kết quả in ra trên một Visual

Basic message box. Sau đây là đoạn mã:



Hình 7-07: GUI đơn giản

```
Private Sub btnAdd_Click()  
    Dim c As New CoCalc  
    MsgBox c.Add(txtNumbl,  
                txtNumb2)  
End Sub
```

Tới lúc này, bạn có một Visual Basic COM server và một Visual Basic COM client chạy rất ăn ý. Bây giờ, ta thử xây dựng một C# client sử dụng COM binary cổ điển như trên.

## 7.4.6 Nhập khẩu Type Library

Bước đầu tiên bạn phải tiến hành trước khi triệu gọi COM server cổ điển (ở đây là VBCOMServerSimple.dll) từ managed code là cho xây dựng một lớp proxy chứa tất cả các thông tin cần thiết dùng tạo RCW. Công cụ chịu trách nhiệm tạo proxy là **tlbimp.exe** (type library importer).

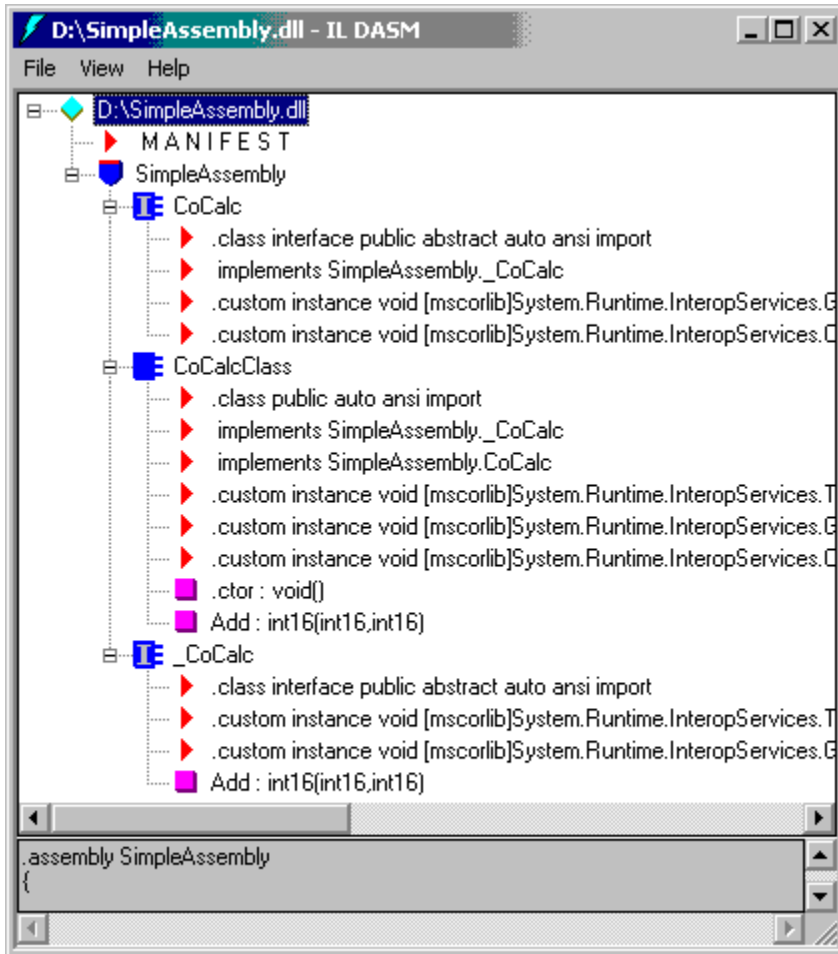
### 7.4.6.1 Trình tiện ích TlbImp.exe

Khi bạn thi hành trình tiện ích này đối với một COM DLL, nó sẽ tra hỏi type library của COM DLL và diễn dịch những thông tin từ đó lên thành dạng thức .NET, chuyển đổi các kiểu dữ liệu COM thành những kiểu dữ liệu .NET như theo bảng 7-04. Một khi bạn cho thi hành **tlbimp.exe** đối với một COM DLL, bạn đơn giản đặt tập tin kết xuất (một RCW) vào thư mục của trình khả thi client sẽ dùng đến nó. (Đây là một bước extra mà bạn phải đi qua khi sử dụng rõ ra trình tiện ích **tlbimp.exe** thay vì dùng khung đối thoại **Add Reference** trên IDE).

Muốn chạy trình tiện ích này từ command line, bạn bắt đầu vào tìm về vị trí của COM binary từ một cửa sổ Command Prompt. Tiếp theo, bạn khai báo tên của COM server và tên của assembly RCW kết xuất (sử dụng flag /out). Sau đây là lệnh:

```
tlbimp VBCOMServerSimple.dll /out:SimpleAssembly.dll
```

Tới đây, coi như bạn đã tạo một proxy mang tên **SimpleAssembly.dll** (một RCW) bạn có thể mở **SimpleAssembly.dll** được kết sinh sử dụng trình tiện ích **ILDasm.exe** (hình 7-08). Chúng tôi sẽ đề cập đến chi tiết IL của assembly này sau trong chương này. Tạm thời, bạn chỉ cần để ý giao diện [default]\_CoCalc cũng như coclass CoCalc được ánh xạ như là tương đương .NET.

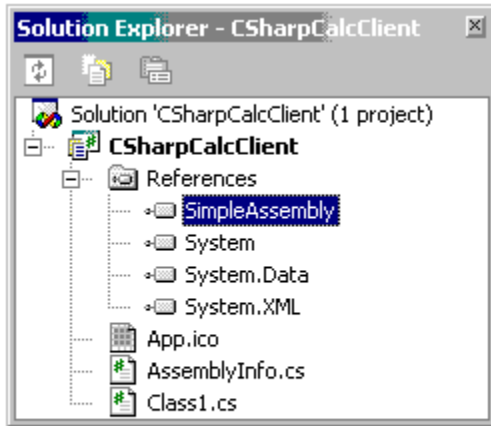


Hình 7-08:: SimpleAssembly.dll được hiển thị bởi ILDasm.exe

### 7.4.7 Qui chiếu tập tin SimpleAssembly.dll

Khi một .NET binary triệu gọi proxy được kết sinh (ở đây là **SimpleAssembly.dll**), yêu cầu này sẽ được chuyển tới lớp COM được gắn liền. Để minh họa tiến trình này sẽ diễn ra một cách đơn giản, ta thử tạo một C# COM Client cho mang tên **CSharpCalcClient**. Bằng cách tạo một C# Console Application mới.

Một khi bạn đã tạo khung sườn của ứng dụng console, bạn cho thêm một qui chiếu về proxy **SimpleAssembly**, như theo hình 7-09.



Hình 7-09: Managed code phải qui chiếu về SimpleAssembly

### 7.4.7.1 Gắn kết sớm về lớp CoCalc COM

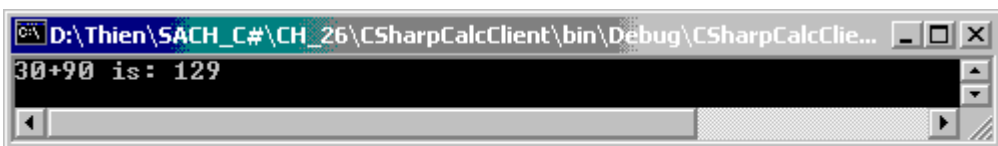
Vì bạn đã thêm trực tiếp qui chiếu về assembly tổng quát, bạn có thể sử dụng early binding (gắn kết sớm). Trong đoạn mã sau đây, đối với C# client, CoCalc chỉ là một kiểu dữ liệu .NET chứa assembly hợp lệ. Thật ra, RCW chặn hứng trước các triệu gọi, và chuyển cho coclass, như theo thí dụ 7-03:

#### Thí dụ 7-03: Gắn kết sớm lớp CoCalc COM

```
*****
using System;
using SimpleAssembly; // qui chiếu namespace chứa proxy

namespace CSharpCalcClient
{
    public class CalClient
    {
        public static int Main(string[] args)
        {
            // Tạo đối tượng calc
            CoCalc c = new CoCalc();
            Console.WriteLine("30+90 is: " + c.Add(30,99));
            Console.ReadLine();
            return 0;
        }
    }
}
*****
```

Hình 7-10 cho thấy kết quả sử dụng gắn kết sớm:



Hình 7-10: Gắn kết sớm với VBCOMServerSimple.dll

Như bạn có thể thấy, tiến trình chuyển đổi trung trực tiếp ra tất cả các thành viên của giao diện [default] của coclass từ một thể hiện đối tượng. Nếu bạn phải qui chiếu rõ ra giao diện \_CoCalc nằm ẩn ở sau, bạn phải viết đoạn mã sau đây, tương đương về mặt logic:

#### ***Thí dụ 7-04: Gắn kết sớm lớp CoCalc COM & sử dụng [default] interface***

```
*****
using System;
using SimpleAssembly;

namespace CSharpCalcClient
{
    public class CalClient
    {
        public static int Main(string[] args)
        {
            // Tạo đối tượng calc
            CoCalc c = new CoCalc();

            // sử dụng tường minh [default] interface
            _CoCalc icalc = c;
            Console.WriteLine(icalc.Add(30, 99));
            Console.ReadLine();
            return 0;
        }
    }
}
*****
```

### ***7.4.7.2 Gắn kết sớm sử dụng Visual Studio .NET***

Bạn để ý là Visual Studio .NET IDE cho phép bạn chọn COM server cổ điển sử dụng tab COM trên khung đối thoại **Add Reference**. Điều này tự động triệu gọi trình tiện ích **tlbimp.exe** và cho trữ assembly mới tạo tại thư mục Debug (hoặc Release) của bạn. Chúng tôi giả sử bạn sẽ sử dụng khung đối thoại **Add Reference** trong phần còn lại của chương này.

### ***7.4.8 Gắn kết trễ về coclass CoCalc***

Trước khi một trình khả thi .NET client có thể triệu gọi các hàm hành sự và thuộc tính của một đối tượng cấu kiện, nó cần phải biết vị chỉ ký ức của các thành viên này. Có hai kỹ thuật khác nhau mà trình client có thể dùng để xác định các vị chỉ này, đó là gắn kết sớm (early binding) và gắn kết trễ (late binding).

Các chương trình chịu gắn kết sớm (early-bound) sẽ xác định được các vị chỉ rất sớm vào lúc biên dịch và metadata sẽ được thêm vào .NET client module. Khi biên dịch, trình

biên dịch sẽ dựa trên type library của component để đưa các vị chỉ của các thành viên của component vào .EXE, như vậy vị chỉ các thành viên này sẽ được truy cập nhanh và không nhập nhằng khi trình khả thi thi hành. Các kỹ thuật COM interoperability mà chúng ta xét đến nay đều dựa vào gắn kết sớm.

Ngược lại, các chương trình chịu gắn kết trễ (late-bound) chỉ có thể xác định vị chỉ các thành viên *vào giờ chót vào lúc chạy, lúc các thành viên này được triệu gọi*. Khi chương trình được thi hành. Thường việc gắn kết trễ được thực hiện thông qua reflection trên C#.

Như bạn còn nhớ trên chương 4, “Tìm hiểu về Attribute và Reflection”, namespace **System.Reflection** cho phép bạn khảo sát bằng lập trình những kiểu dữ liệu nào đó trong một assembly vào lúc chạy. Trên COM, chức năng này cũng được hỗ trợ thông qua việc sử dụng một lô giao diện chuẩn (chẳng hạn ITypeLib, ITypeInfo, v.v..). Khi một client gắn kết với một thành viên vào lúc chạy (thay vì vào lúc biên dịch), ta bảo là client sử dụng gắn kết trễ (late binding).

Nhìn chung, có thể bạn khoái sử dụng kỹ thuật early binding mà ta đã xem qua trước đây. Tuy nhiên, cũng có lúc ta cần sử dụng late binding đối với một coclass COM. Thí dụ, một vài COM server do quá khứ để lại (legacy) được xây dựng thế nào đó mà nó lại không cung cấp bất cứ thông tin nào liên quan đến kiểu dữ liệu. Như vậy trong trường hợp này, rõ ràng là bạn không thể chạy trình tiện ích **tlbimp.exe**. Do đó, bạn có thể truy cập các kiểu dữ liệu COM cổ điển bằng cách sử dụng các dịch vụ .NET Reflection.

Tiến trình gắn kết trễ bắt đầu với một client nhận được giao diện **IDispatch** từ một coclass nào đó. Giao diện COM chuẩn này định nghĩa tổng cộng 4 hàm hành sự, mà bạn chỉ quan tâm trong lúc này 2 hàm mà thôi. Hàm thứ nhất là **GetIDsOfNames()** cho phép client nhận được một trị số (được gọi là DISPID) dùng nhận diện hàm hành sự mà phía client muốn triệu gọi.

Trên COM IDL, DISPID của một thành viên sẽ được gán thông qua attribute [id]. Nếu bạn quan sát đoạn mã IDL được kết sinh bởi Visual Basic (sử dụng OLE/COM Object Viewer), bạn sẽ thấy DISPID của hàm hành sự **Add()** được gán như sau:

```
[id(0x60030000)]
HRESULT Add([in] short x, [in] short y, [out, retval] short* );
```

Đây là trị số mà **GetIDsOfNames()** trả về. Một khi có trong tay trị này, client có thể triệu gọi hàm kế tiếp là **Invoke()**. Hàm này chấp nhận một số thông số, một trong số này là DISPID do **GetIDsOfNames()** trả về.

Ngoài ra, hàm hành sự **Invoke()** cần đến một bản dãy kiểu dữ liệu COM VARIANT tượng trưng các thông số sẽ được trao qua hàm. Trong trường hợp hàm hành sự **Add()** thì



bản đây này chứa hai trị nào đó kiểu short. Thông số cuối cùng của **Invoke()** là một VARIANT khác lo cầm giữ trị trả về (kiểu short) của hàm hành sự được triệu gọi hàm.

Mặc dù một client sử dụng late binding không trực tiếp sử dụng giao diện **IDispatch**, chức năng tổng quát này cũng được diễn ra trong namespace **System.Reflection**.

Khi bạn gắn kết trễ một đối tượng COM trên một chương trình C#, bạn không cần tạo một proxy RCW đối với COM component. Thay vào đó, bạn triệu gọi hàm **GetTypeFromProgID()** trên lớp **Type** để hiển lộ một đối tượng tượng trưng cho kiểu dữ liệu đối tượng COM. Lớp **Type** là thành viên của namespace **System.Runtime.InteropServices**. Đoạn mã sau đây ta cấu hình hóa một đối tượng Type qui chiếu về một coclass CoCalc:

```
using System.Runtime.InteropServices;
Type calcObj;
calcObj = Type.GetTypeFromProgID("VBCOMServerSimple.CoCalc");
```

Triệu gọi hàm **GetTypeFromProgID** yêu cầu NET Framework mở COM DLL và tìm lấy thông tin kiểu dữ liệu cần thiết đối với đối tượng được khai báo. Đây cũng tương đương với việc triệu gọi hàm **GetType** trong chương 4, “Tìm hiểu về Attribute và Reflection”.

Một khi bạn đã có một đối tượng **Type** (calcObj) gói ghém type information của đối tượng COM, bạn dùng đối tượng calcObj này để tạo một thể hiện late-bound của bản thân đối tượng COM, thông qua hàm hành sự **CreateInstance()** thuộc lớp **Activator**, như theo dòng lệnh sau đây:

```
object calcDisp = Activator.CreateInstance(calcObj);
```

Tới đây, đoạn mã C# có một qui chiếu gắn kết trễ về một thể hiện của coclass COM. Tiếp theo, bạn tạo một bản đây để giữ các đối mục của hàm hành sự COM (ở đây là Add):

```
object[] addArgs = {100, 34};
```

Rất tiếc là bạn không thể triệu gọi hàm trực tiếp trên qui chiếu object. Thay vào đó, muốn nói chuyện với đối tượng COM, bạn chỉ có thể dựa vào hàm hành sự **InvokeMember** của đối tượng **Type** (ở đây là calcObj), trao qua một qui chiếu về đối tượng COM (ở đây là calcDisp), kèm theo tên hàm hành sự COM (ở đây là Add chẳng hạn) và một bản đây object bao gồm bất cứ đối mục nào của hàm hành sự COM (ở đây là addArgs). Ta có lệnh sau đây:

```
sum = calcObj.InvokeMember(
    "Add", // hàm cần triệu gọi
```

```

BindingFlags.InvokeMethod, // gắn kết thể nào
null,                      // binder
calcDisp,                  // đối tượng COM
addArgs);                  // đối mục của hàm hành sự

```

Để minh họa những điều vừa kể trên, sau đây là thí dụ 7-05 sử dụng late binding để triệu gọi hàm **Add()**. Bạn để ý là ứng dụng này không qui chiếu về assembly (chẳng hạn **SimpleAssembly** như thí dụ đi trước) nên sẽ không cần đến trình tiện ích **tlbimp.exe**:

### ***Thí dụ 7-05: Late binding dùng triệu gọi hàm Add() thuộc CoCalc COM***

```

*****
using System;
using System.Reflection;
using System.Runtime.InteropServices;

namespace CSharpLateBindCalcClient
{
    public class LateBinder
    {
        public static int Main(string[] args)
        {
            // Trước tiên, đi lấy qui chiếu IDispatch từ coclass
            Type calcObj =
                Type.GetTypeFromProgID("VBCOMServerSimple.CoCalc");

            object calcDisp = Activator.CreateInstance(calcObj);

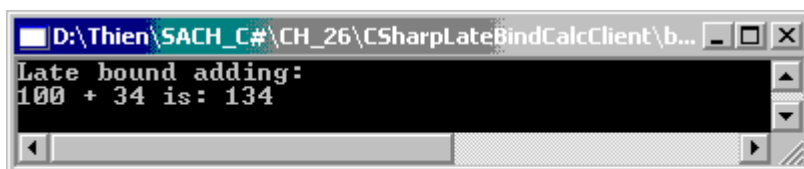
            // Tạo một bản dãy các đối mục của hàm Add()
            object[] addArgs = {100, 34};

            // triệu gọi hàm Add() và tính tổng cộng
            object sum = null;
            sum = calcObj.InvokeMember("Add",
                BindingFlags.InvokeMethod, null, calcDisp, addArgs);
            Console.WriteLine("Late bound adding:\n100 + 34 is: {0}",
                sum);
            Console.ReadLine();

            return 0;
        }
    }
}
*****

```

Hình 7-11 cho thấy kết xuất:



**Hình 7-11: Gắn kết trễ về VBCOMServerSimple**

**Bạn để ý:** Tới đây bạn đã biết late binding là gì rồi. Tuy nhiên, bạn phải chú ý đến vài điều bất lợi khi dùng gắn kết trễ. Thứ nhất, late binding có thể rất nguy hiểm. Khi bạn sử dụng gắn kết sớm, trình biên dịch còn có khả năng tham khảo type library của COM component để bảo đảm là tất cả các hàm hành sự mà bạn triệu gọi đối với các đối tượng COM thật sự hiện hữu. Còn với late binding, không gì ngăn cản việc khở sai trong triệu gọi hàm **InvokeMember()** gây ra một sai lầm. Thứ hai, late binding có thể chạy rất chậm. Mỗi lần bạn sử dụng **InvokeMember()** đối với một qui chiếu **object**, CLR phải đi dò tìm hàm thành viên mong muốn trên function library của coclass COM. Cuối cùng, các lệnh late binding rất khó viết và mất thời giờ. Vì bạn không có một qui chiếu về type library của COM component, nên Intellisense của IDE không thể giúp bạn nên có thể bạn thêm những sai lầm trong đoạn mã và chỉ phát hiện vào lúc chạy.

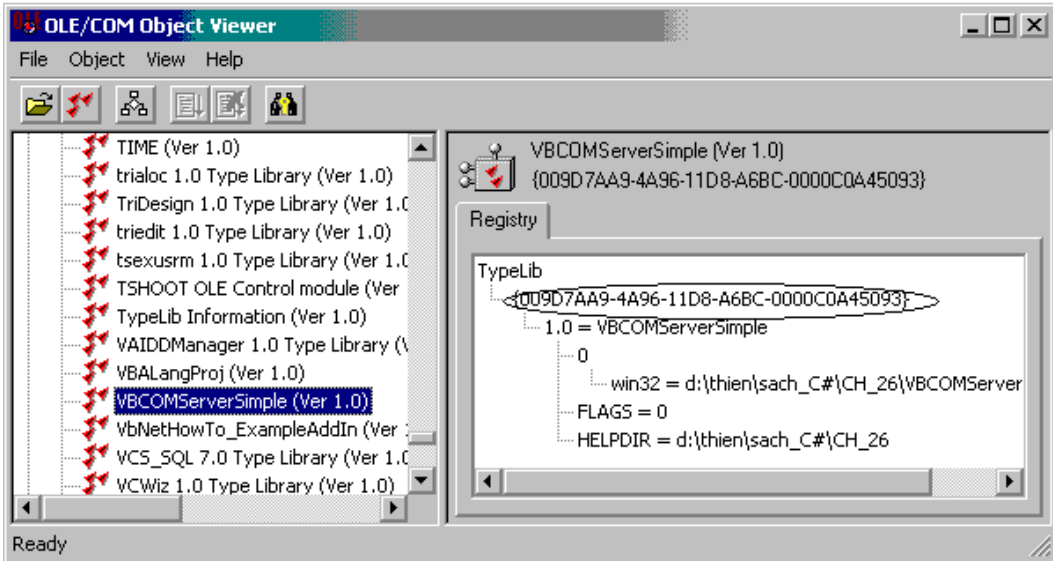
## 7.4.9 Khảo sát assembly được kết sinh

Bây giờ bạn đã biết làm thế nào hiện dịch (activate) một kiểu dữ liệu COM từ managed code, ta thử nhìn xem vài chi tiết đặc trưng. Bạn cho nạp **SimpleAssembly.dll** (do trình tiện ích **tlbimp.exe** tạo ra) vào **ILDasm.exe** và cho double click lên Manifest để cho hiện lên khung đối thoại Manifest. Giống như bất cứ assembly nào, bạn sẽ thấy một qui chiếu ngoại lai về mscorlib.dll (core .NET class library), theo sau là thông tin phiên bản cần thiết.

Thông tin quan trọng là một số attribute .NET. Khi bạn quan sát manifest, bạn sẽ thấy những qui chiếu về **GuidAttribute** và **ImportedFromTypeLibAttribute**. Nếu bạn nhìn vào trị của **ImportedFromTypeLibAttribute** thì bạn thấy lối tìm về của COM server cổ điển. Điều này minh họa một điểm quan trọng. Nếu COM Server được tái định cư đâu đó (hoặc đổi tên) trên máy tính đích, thì bạn phải cho kết sinh lại assembly. Còn **GuidAttribute** cho biết trị của GUID được gán cho type library (LIBID). Nếu bạn dùng OLE/COM Object Viewer, bạn sẽ thấy khớp với trị số của **GuidAttribute** attribute. Xem hình 7-12.

Tiếp theo là bạn có bản thân CoCalc class type. Ngoài hàm hành sự **Add()**, CoCalc phải được cung cấp với một hàm constructor mặc nhiên. Đây có nghĩa là RCW có trung coclass thô như là một .NET type, nên cần một hàm constructor để khởi động lớp coclass này. Nếu bạn xem hình 7-08, bạn thấy những chỉ thị IL khác nhau đánh dấu lớp cơ bản trừu tượng CoCalc và các giao diện thi công (**\_CoCalc**).

```
.class public auto ansi import CoCalcClass
    extends [mscorlib]System.Object
    implements SimpleAssembly._CoCalc,
        SimpleAssembly.CoCalc
{
    . . .
} // end of class CoCalcClass
```



Hình 7-12: Kiểm tra số GUID

Bạn có thể khảo sát xa hơn việc chuyển đổi COM qua .NET, sau đây là một phiên bản nhật tu hàm Main() khảo sát kiểu dữ liệu COM sử dụng các thành viên của **System.Object** và **System.Type**, xem thí dụ 7-06:

### Thí dụ 7-06: Khảo sát COM type

```

*****
using System;
using SimpleAssembly;

namespace CSharpCalcClient
{
    public class CalClient
    {
        public static int Main(string[] args)
        {
            CoCalc c = new CoCalc();
            Console.WriteLine("30+90 is: " + c.Add(30,99));

            // COM type chịu hỗ trợ System.Object.ToString()
            Console.WriteLine("->CoCalc to string: {0}", c.ToString());

            // Trích ra một vài thông tin của COM
            Type t = c.GetType();
            Console.WriteLine("->COM class?: {0}", t.IsCOMObject);
            Console.WriteLine("->Full Name?: {0}", t.FullName);
            Console.WriteLine("->CLSID?: {0}", t.GUID.ToString());
            Console.WriteLine("->Is it a interface?: {0}",
                               t.IsInterface);

            Console.ReadLine();
        }
    }
}

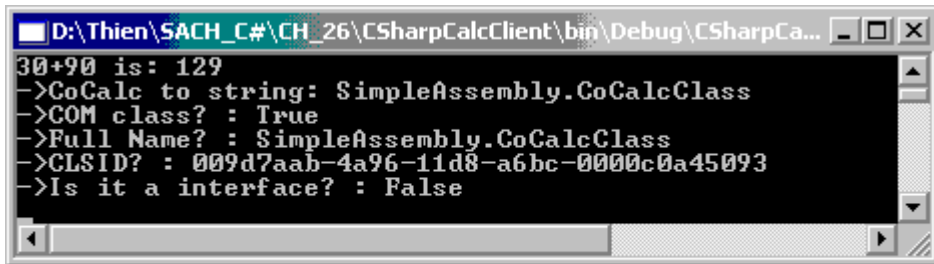
```

```

        return 0;
    }
}
}
*****

```

Hình 7-13 cho thấy kết xuất



Hình 7-13: Khảo nghiệm COM type của Bạn

Tới đây, hy vọng bạn đã biết cách sử dụng một COM server trên một managed code. Bây giờ, ta thử xem cách sử dụng ActiveX control trên .NET.

## 7.5 Sử dụng ActiveX control trên .NET

ActiveX control là một loại COM component rất đặc biệt chịu hỗ trợ một lô giao diện đặc biệt cung cấp những khía cạnh đồ họa và có thể được lồng vào một biểu mẫu. Như bạn có thể đã biết, ActiveX control mang phần đuôi là .ocx. Khi Microsoft phát triển chuẩn OCX, cho phép lập trình viên xây dựng những ActiveX control trên Visual Basic và đem sử dụng chúng với C++ (hoặc ngược lại) thì cuộc cách mạng ActiveX control bắt đầu. Sau nhiều năm, hàng ngàn ô control OCX được triển khai, bán qua thị trường và được sử dụng. Các ô control này tương đối nhỏ, dễ sử dụng và là một thí dụ về việc tái sử dụng những binary.

Giống như việc bạn nhập khẩu những component COM chuẩn dùng trong các dự án .NET như trong các mục đi trước, bạn cũng có thể nhập khẩu dễ dàng ActiveX control vào .NET, mặc dù chuẩn COM binary và .NET binary không hề tương thích. Visual Studio .NET có khả năng nhập khẩu ActiveX control, và Microsoft cũng chế ra một trình tiện ích command line mang tên **AxImp.exe** cho phép bạn nhập khẩu ActiveX control. **AxImp.exe** sẽ tạo những assembly cần thiết đối với ActiveX control để có thể dùng trên các ứng dụng .NET.

## 7.5.1 Tạo một ActiveX control

Để minh họa khả năng sử dụng ActiveX control cổ điển trên một ứng dụng .NET, trước tiên bạn cho triển khai một máy tính bỏ túi đơn giản gồm 4 chức năng (cộng, trừ, nhân và chia) như là một ActiveX control rồi sau đó cho triệu gọi ActiveX control này trong một ứng dụng .NET. Bạn sẽ tạo ActiveX control này trên Visual Basic 6.0.

Một khi ô control này được tạo xong chạy trên môi trường Windows, bạn sẽ cho chép nó lên môi trường .NET, cho nó đăng ký lên hệ thống rồi cho nhập khẩu vào một ứng dụng Windows Forms.

Muốn tạo ô control, bạn cho mở Visual Basic 6.0, rồi chọn ActiveX control như là kiểu dự án. Bạn cho biểu mẫu thu nhỏ lại càng nhỏ càng tốt, vì ô control này không cần có một giao diện người sử dụng. Bạn right-click lên **UserControl**, chọn click **Properties**, rồi đổi tên thành **Calculator** trên cửa sổ **Properties**. Tiếp theo, bạn right-click lên Project1 rồi chọn click mục Project1 Properties để cho khung đối thoại Project Properties hiện lên. Đổi tên Project1 thành **CalcControl**. Lập tức cho **Save** dự án và đặt tên tập tin và dự án là **CalcControl** vào thư mục bạn muốn.

Bây giờ bạn có thể thêm 4 hàm máy tính. Bạn right-click lên biểu mẫu CalcControl, chọn View Code (hoặc ra lệnh View | Code) bạn cho khổ đoạn mã Visual Basic như theo Thí dụ 7-07 sau đây:

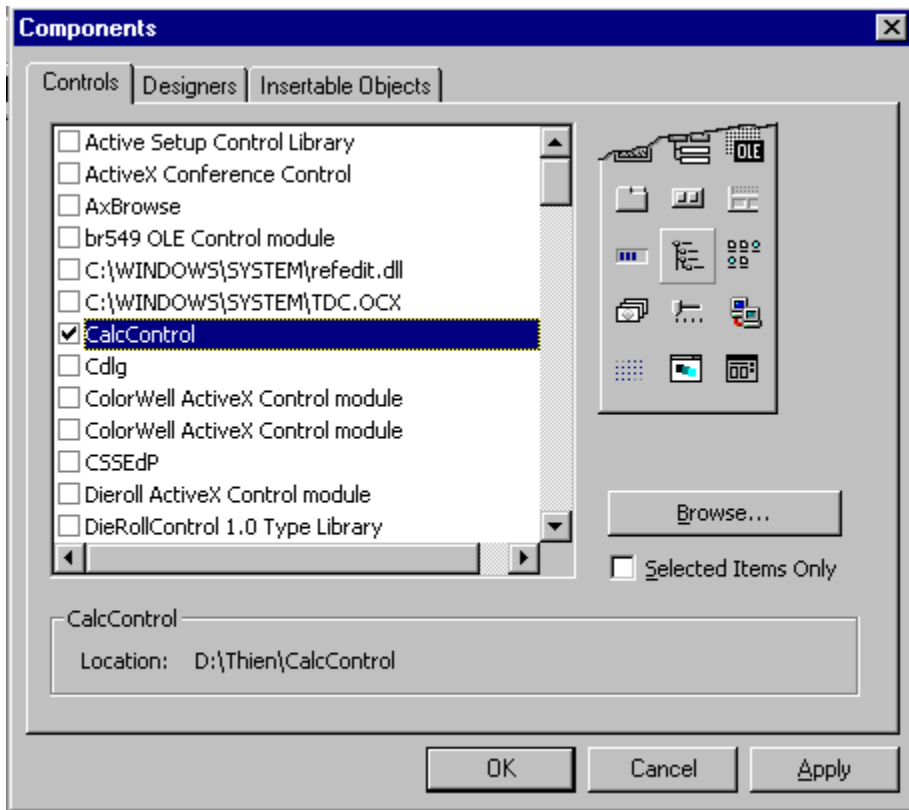
### **Thí dụ 7-07: Thi công ActiveX control CalcControl**

```
*****
Public Function Add (left As Double, right As Double) As Double
    Add = left + right
End Function
Public Function Subtract (left As Double, right As Double) As Double
    Subtract = left - right
End Function
Public Function Multiply (left As Double, right As Double) As Double
    Multiply = left * right
End Function
Public Function Divide (left As Double, right As Double) As Double
    Divide = left / right
End Function
*****
```

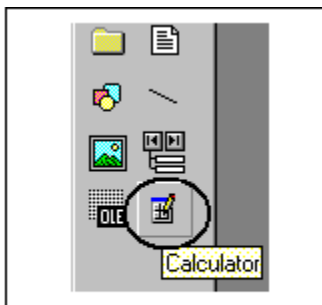
Bạn cho biên dịch đoạn mã này bằng cách ra lệnh **File | Make CalcControl.ocx** trên trình đơn Visual Basic 6.0.

Tiếp theo, bạn mở một dự án thứ hai trên Visual Basic nhưng lần này là **Standard EXE** và cho đặt tên biểu mẫu là **TestForm** và tên dự án là **CalcTest**, rồi cho save dự án dưới cái tên **CalcTest**.

Bạn thêm ActiveX control như là component, bằng cách ấn tổ hợp phím <Ctrl+T> để cho hiện lên khung đối thoại Components (hình 7-14), rồi chọn CalcControl từ Tab Controls như theo hình 7-14. Việc này đưa những ô control mới vào toolbox. (hình 7-15)



**Hình 7-14: Thêm CalcControl vào VB6 Toolbox**

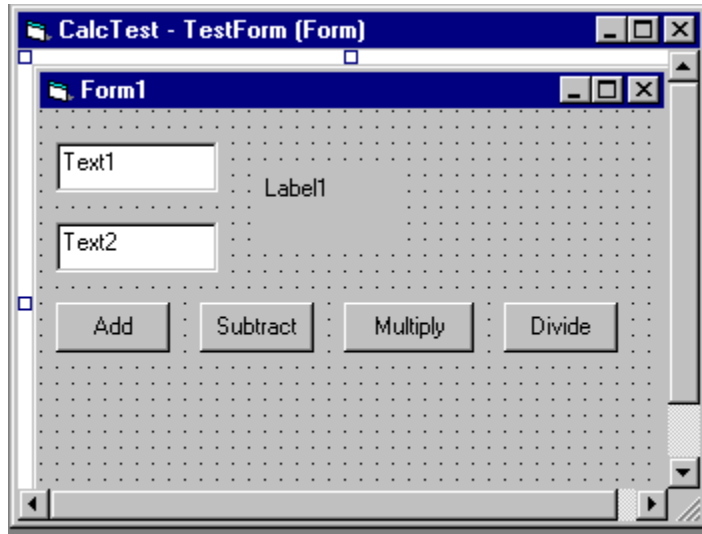


**Hình 7-15: CalcControl trên VB6 toolbox**

Bạn cho lỗi ô control CalcControl mới này vào biểu mẫu TestForm và cho đặt tên CalControl. Bạn để ý là ô control này không hiện lên vì nó không giả sử có người sử dụng. Bạn thêm hai text box (Text1 và Text2), 4 button (btnAdd, btnSubtract, btnMultiply, và btnDivide) và một label (Label1) như theo hình 7-16.

Vấn đề còn lại là viết những hàm thụ lý tình huống các nút bị ấn xuống. Mỗi lần nút bị ấn xuống bạn đi lấy trị trong hai text box, ép nó về kiểu double (thông qua hàm CDBl()), triệu gọi hàm CalcControl rồi in ra kết quả trên label control Label1. Thí dụ 7-08 cho thấy toàn bộ đoạn mã viết

theo Visual Basic 6.0.



Hình 7-16: Xây dựng TestForm user interface

:

**Thí dụ 7-08: Sử dụng CalcControl trên một chương trình Visual Basic**

```
*****
Private Sub btnAdd_Click()
    Label1.Caption = CalcControl.Add(CDbl(Text1.Text), _
                                     CDbl(Text2.Text))
End Sub

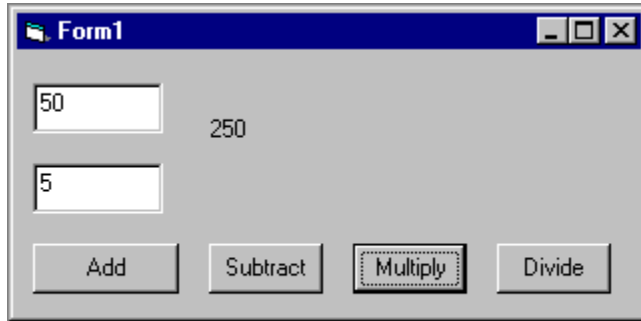
Private Sub btnDivide_Click()
    Label1.Caption = CalcControl.Divide(CDbl(Text1.Text), _
                                         CDbl(Text2.Text))
End Sub

Private Sub btnMultiply_Click()
    Label1.Caption = CalcControl.Multiply(CDbl(Text1.Text), _
                                           CDbl(Text2.Text))
End Sub

Private Sub btnSubtract_Click()
    Label1.Caption = CalcControl.Subtract(CDbl(Text1.Text), _
                                           CDbl(Text2.Text))
End Sub
*****
```

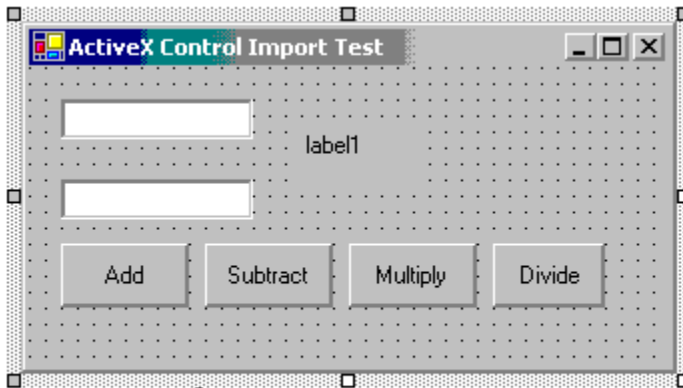
Hình 7-17 cho thấy kết quả chạy chương trình:





Hình 7-17: Kết xuất trình CalcTest

## 7.5.2 Nhập khẩu một ActiveX control vào .NET



Hình 7-18: Biểu mẫu Windows Forms dùng thử trải nghiệm CalcControl ActiveX control

Bây giờ bạn biết là ActiveX control **CalcControl.ocx** chạy tốt, bạn có thể sao tập tin CalcControl.ocx qua môi trường .NET. Một khi sao qua xong bạn cho đăng ký tập tin này sử dụng trình tiện ích Regsvr32 như sau:

```
Regsvr32 CalcControl.ocx
```

Nếu việc đăng ký thành công, bạn có thể xây dựng một chương trình trải nghiệm chạy trên .NET.

Bạn bắt đầu tạo một ứng dụng Windows Forms trên Visual Studio .NET, cho mang tên InteropTest và thiết kế giống như hình 7-16. Hình 7-18 cho thấy biểu mẫu thực sự:

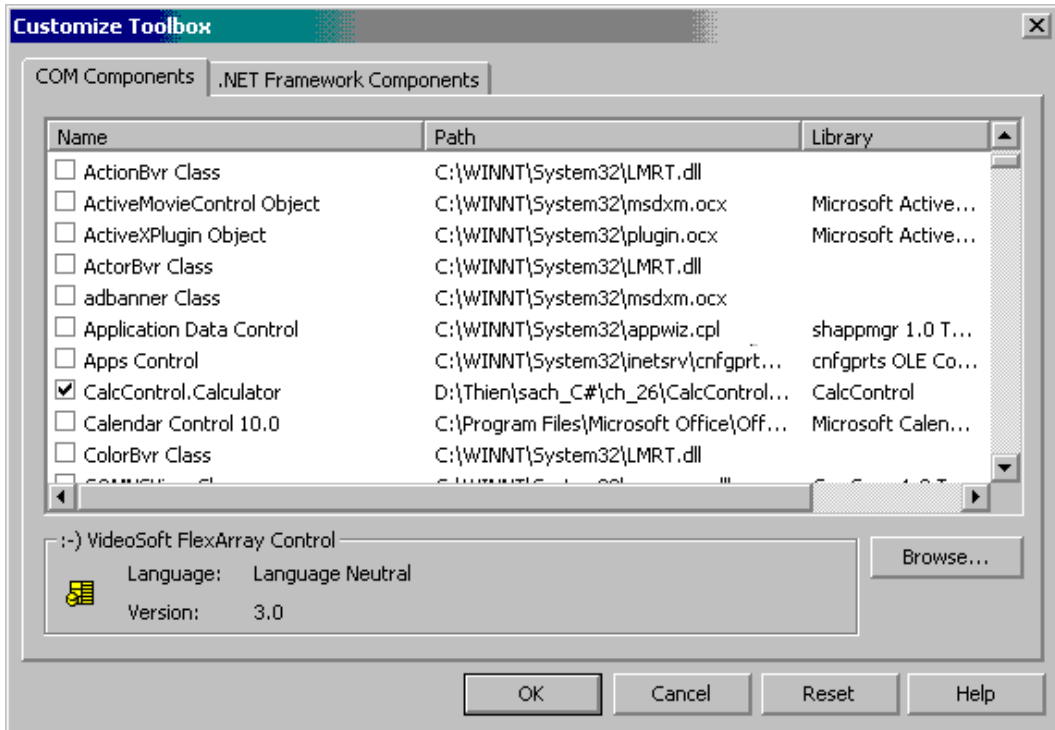
### 7.5.2.1 Nhập khẩu một ô control

Có hai cách để nhập khẩu một ActiveX control vào một môi trường triển khai Visual Studio .NET:

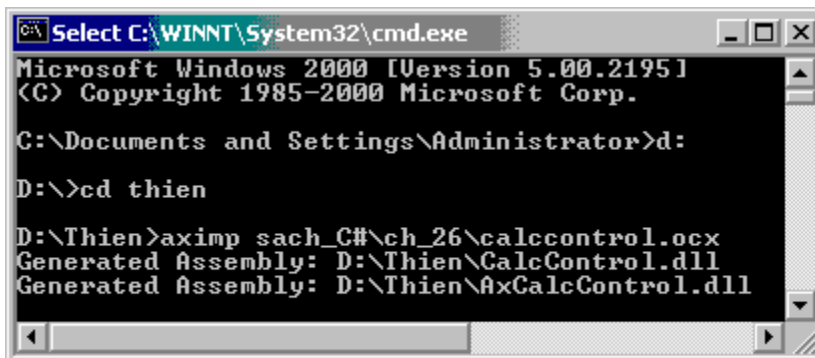
- Bạn có thể dùng các công cụ của Visual Studio .NET, hoặc

- Bạn sử dụng trình tiện ích **Aximp.exe** đi kèm theo .NET SDK

Muốn sử dụng Visual Studio .NET, bạn ra lệnh **Tools | Customize Toolbox** từ trình đơn chính của IDE. Khi khung đối thoại **Customize Toolbox | Tab COM components** hiện lên, bạn cho ON ô check **CalcControl.Calculator** mà bạn vừa đăng ký xong. Xem hình 7-19.



Hình 7-19: Nhập khẩu CalcControl ActiveX Control



Hình 7-20: Cho chạy AxImp.exe

Vì CalcControl được đăng ký trên máy tính .NET, Customi ze Toolbox có khả năng tìm thấy nó. Khi bạn chọn ô control này từ khung đối thoại hình 7-19 kể trên, thì ô control này được

nhập khẩu vào ứng dụng của bạn. Visual Studio .NET sẽ lo mọi chi tiết.

Cách thứ hai là bạn cho mở command line rồi sử dụng trình tiện ích **aximp .exe** để nhập khẩu bằng tay ô control này, như theo hình 7-20:

Trình tiện ích aximp.exe chỉ nhận một đối mục là ActiveX control mà bạn muốn nhập khẩu (CalcControl.ocx). Nó sẽ tạo ra 3 tập tin:

- **AxCalcControl.dll**: một .NET Windows control
- **CalcControl.dll**: một proxy .NET class library
- **AxCalcControl.pdb**: một tập tin debug.

### 7.5.2.2 Thêm một ô control vào Visual Studio toolbox

Một khi xong việc này, bạn có thể trở về cửa sổ Customize Toolbox, nhưng lần này chọn **tab .NET Framework Components**. Giờ đây, bạn có thể rà qua tìm nơi cư trú của .NET Windows control *AxCalcControl.dll* được kết sinh và cho nhập khẩu tập tin này vào toolbox. Một khi được nhập khẩu, ô control hiện lên trên trình đơn toolbox.

Bây giờ bạn có thể lôi thả ô control này lên biểu mẫu Windows Form, mang tên mặc nhiên là axCalculator1. Bạn có thể sử dụng các hàm của ô control này giống như trên thí dụ Visual Basic 6.0.

Bạn thêm các hàm thụ lý tình huống đối với mỗi nút btnAdd, btnMultiply, btnSubtract và btnDivide. Thí dụ 7-09 cho thấy đoạn mã của các hàm tài khoản tình huống:

#### Thí dụ 7-09: Thi công các hàm thụ lý tình huống dùng trực nghiệm Windows Form

```
*****
private void btnAdd_Click(object sender, System.EventArgs e)
{
    double left = double.Parse(textBox1.Text);
    double right = double.Parse(textBox2.Text);
    label1.Text = axCalculator1.Add(ref left, ref right).ToString();
}

private void btnSubtract_Click(object sender, System.EventArgs e)
{
    double left = double.Parse(textBox1.Text);
    double right = double.Parse(textBox2.Text);
    label1.Text = axCalculator1.Subtract(ref left, ref right).ToString();
}

private void btnMultiply_Click(object sender, System.EventArgs e)
{
    double left = double.Parse(textBox1.Text);
    double right = double.Parse(textBox2.Text);
    label1.Text = axCalculator1.Multiply(ref left, ref right).ToString();
}
```

```

        label1.Text = axCalculator1.Multiply(ref left,ref right).ToString();
    }

    private void btnDivide_Click(object sender, System.EventArgs e)
    {
        double left = double.Parse(textBox1.Text);
        double right = double.Parse(textBox2.Text);
        label1.Text = axCalculator1.Divide(ref left,ref right).ToString();
    }

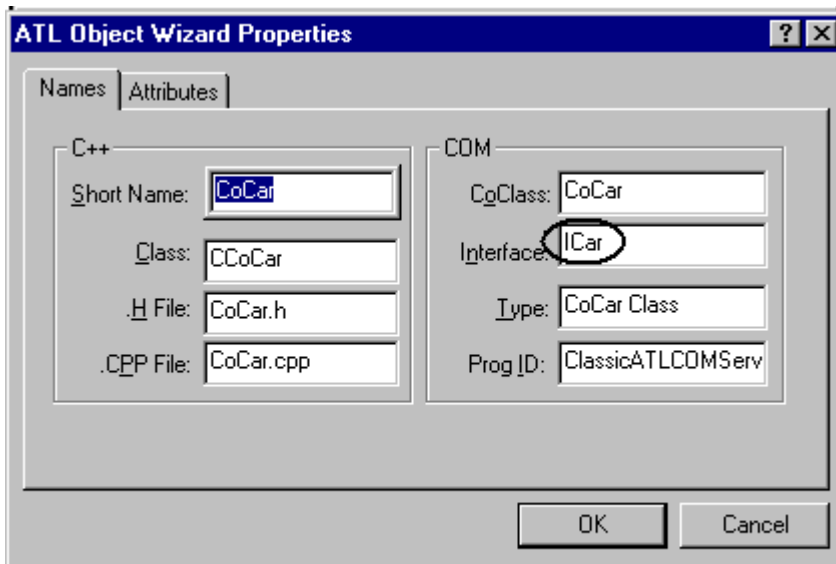
```

Các trị vùng mục tin text được chuyển đổi thành double sử dụng hàm hành sự static **double.Parse()**, rồi chuyển các trị này cho máy tính bỏ túi. Kết quả được ép về cho một chuỗi rồi cho hiển thị lên label. Kết quả cũng giống như trên hình 7-17.

Bước kế tiếp là tìm hiểu sâu tiến trình chuyển đổi và một số qui tắc đặc trưng mà trình tiện ích **tlbimp.exe** sử dụng trong việc chuyển đổi.

## 7.6 Xây dựng một ATL Test Server

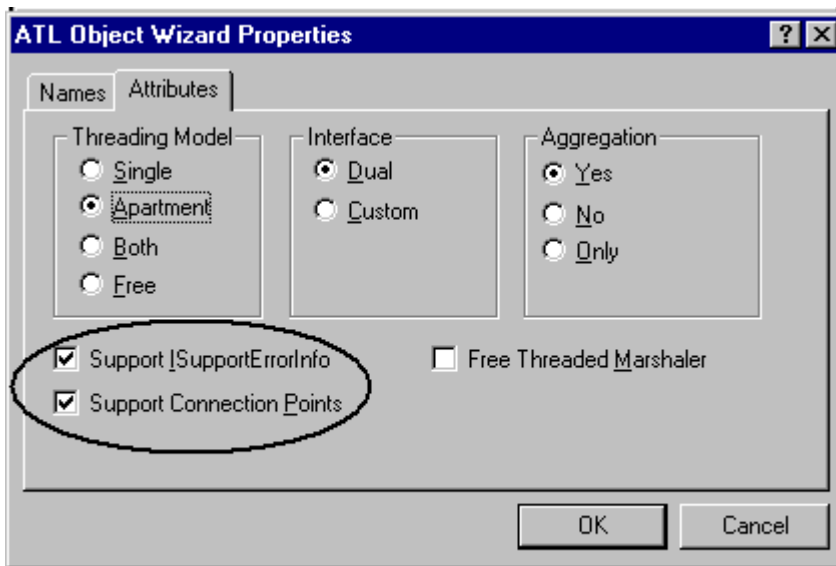
Muốn thật sự hiểu tiến trình chuyển đổi (conversion process), bạn nên tập trung xây dựng một ATL COM server. Đây là dịp tốt bạn làm quen với IDL. Bạn sẽ hiểu sâu làm thế nào COM SAFEARRAY, BSTR, enum, coclass và interface sẽ ánh xạ với những đối tượng tương đương trên .NET. Tập sách này không phải là một tutorial đối với ATL. Do đó, phần mục này sẽ mô tả từng bước một tiến trình xây dựng một ATL COM Server.



Hình 7-21: Đặt tên cho ATL coclass mới

Để bắt đầu, bạn cho mở **Visual Studio C++ 6.0** và chọn một ATL project workspace cho đặt tên **ClassicATLCOMServer**, bằng cách ra lệnh **New | Project | ATL COM Wizard**. (Trong khi bạn có thể dùng ATL 4.0 và Visual Studio .NET, ta thử chơi với ATL 3.0 khá phổ biến). Kế tiếp, cho mở **ATL Object Wizard** (bằng cách ra lệnh **Insert | New ATL Object | Simple Object...**). Bạn sử dụng **Tab Name** của khung đối thoại **ATL Object Wizard Properties** (hình 7-21 trang trước) để đổi tên [default] interface thành **ICar**, (thay vì **ICoCar**).

Tiếp theo bạn cho về ON các ô duyệt **ISupportErrorInfo** và **SupportConnectionPoints** trên **Tab Attributes**, vì coclass bạn đang xây dựng có khả năng gọi đi COM error và phát pháo các tiến hành trở về cho .NET Client (hình 7-22). Còn các mục khác để yên chấp nhận trị mặc nhiên. Cuối cùng bạn click OK và cho biên dịch lần đầu tiên.

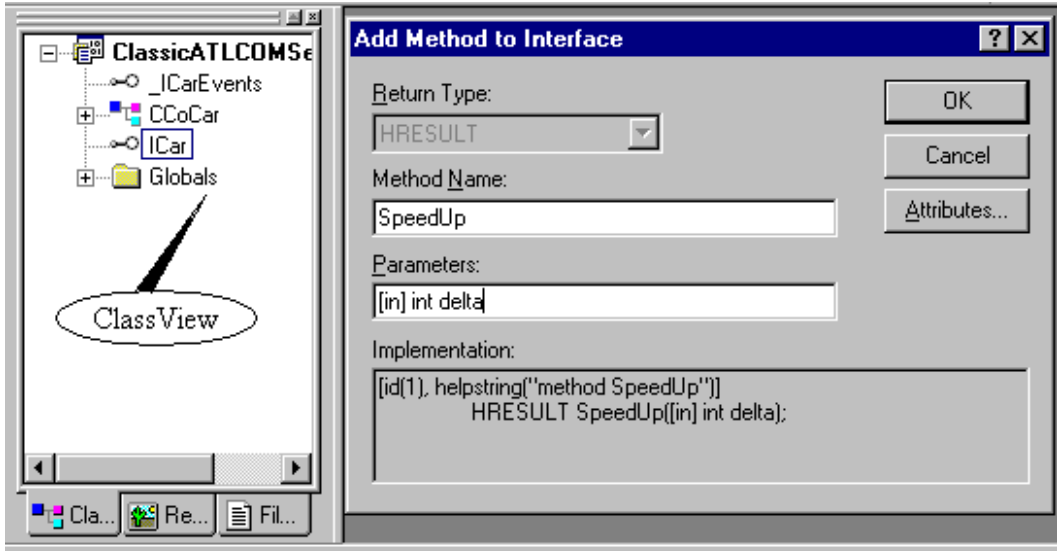


Hình 7-22: Thêm hỗ trợ đối với COM error và COM event

## 7.6.1 “Thêm mắm thêm muối” cho [default] COM Interface

Kế tiếp bạn cần thêm một vài thành viên ban đầu cho giao diện [default] **ICar**. Từ cửa sổ ClassView, bạn right-click lên giao diện **ICar**, để cho hiện lên trình đơn shortcut rồi bạn chọn click mục **Add Method**. Khung đối thoại **Add Method** hiện lên. Bạn cho tạo hàm hành sự **SpeedUp()** với một thông số kiểu dữ liệu **int**. (hình 7-23).

Hàm hành sự thứ hai, mang tên **GetCurSpeed()**, trả về một con trỏ do thông số [out, retval]. Một khi bạn đã thêm mỗi thành viên, định nghĩa ban đầu IDL của bạn sẽ giống như sau:



**Hình 7-23: Định nghĩa các thông số cho hàm hành sự sử dụng IDL attribute**

```
// ClassicATLCOMServer.idl: IDL source for ClassicATLCOMServer.dll
//
// This file will be processed by the MIDL tool to
// produce the type library (ClassicATLCOMServer.tlb) and marshalling
// code.

import "oaidl.idl";
import "ocidl.idl";

[
    object,
    uuid(ADB431A3-4C28-11D8-A6BC-0000C0A45093),
    dual,
    helpstring("ICar Interface"),
    pointer_default(unique)
]
interface ICar: IDispatch
{
    [id(1), helpstring("method SpeedUp")]
    HRESULT SpeedUp([in] int delta);

    [id(2), helpstring("method GetCurSpeed")]
    HRESULT GetCurSpeed([out, retval] int* currSp);
};
```

Việc thi công đoạn mã hỗ trợ coclass này rất đơn giản. Trước tiên, bạn thêm một biến thành viên kiểu dữ liệu int (mang tên curSpeed) vào lớp **CoCar** mới, bằng cách right-click lên **CCoCar** trên ClassView, rồi chọn click **Add Member Variable** rồi thêm biến thành viên **int curSpeed**. Bạn cho gán biến này về trị zero trong hàm constructor **CoCar()** và thi công mỗi hàm hành sự như sau:

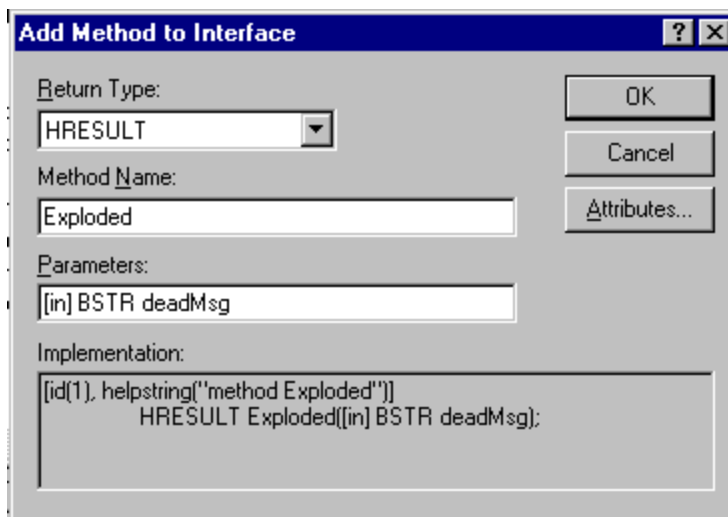
```
STDMETHODIMP CCoCar::SpeedUp(int delta)
{
    // Thêm delta cho curSpeed
    curSpeed += delta;
    return S_OK;
}

STDMETHODIMP CCoCar::GetCurSpeed(int *currSp)
{
    // trả về current speed
    *currSp = curSpeed;
    return S_OK;
}
```

Để bảo đảm là bạn không đưa bug, bạn cho biên dịch server này. Nếu không gì sai, thế là tốt rồi.

## 7.6.2 Cho phát pháo một COM Event

Bây giờ bạn thêm chức năng cho lớp **CoCar** để có thể phát pháo một tình huống COM. Bước đầu tiên là thêm một hàm hành sự cho outbound interface của bạn tượng trưng cho những hàm hành sự mà coclass sẽ triệu gọi đối với client sink (bạn nhớ lại outbound interface được định nghĩa bởi COM server nhưng lại được thi công bởi client). Trên ATL, outbound interface (giao diện gắn kết ra ngoài) được liệt kê trên đầu



Hình 7-24: Thêm một hàm thụ lý tình huống

ClassView được đánh dấu bởi dấu “\_” làm prefix và “Events” làm suffix. Trong trường hợp của chúng ta đó là **\_ICarEvents**. Bạn right-click **\_ICarEvents** để cho hiện lên trình đơn shortcut, bạn chọn click **Add Method**. Khung đối thoại **Add Method to Interface** hiện lên. Bạn cho thêm một tình huống xe nổ tung mang tên **Exploded**, gọi cho client một thông số kiểu BSTR (hình 7-24).

```
[
    uuid(ADB431A5-4C28-11D8-A6BC-0000C0A45093),
    helpstring("_ICarEvents Interface")
]
dispinterface _ICarEvents
{
    properties:
    methods:
        [id(1), helpstring("method Exploded")]
        HRESULT Exploded([in] BSTR deadMsg);
};
```

Một khi bạn đưa vào tình huống **Exploded** này vào giao diện outbound, bạn cho biên dịch lại dự án để nhật tu thông tin COM type. Kế tiếp, bạn cần xây dựng một event proxy, mà ATL framework sẽ dùng đến để phát pháo tình huống **Exploded** đối với bất cứ client nào được kết nối. Muốn thế, bạn right-click lên nhánh CoCar từ cửa sổ ClassView và chọn click mục **Implement Connection Point**. Khi khung đối thoại hiện lên, bạn cho ON ô duyệt mang tên giao diện tình huống (\_ICarEvents) rồi click <OK>.

Công cụ này phản ứng bằng cách nhật tu CoCar trong nhiều cách. Trước tiên, bạn sẽ thấy CONNECTION\_MAP (được liệt kê trong tập tin CoCar.h) đã được nhật tu theo mục vào mới. Bạn nên biết công cụ này có một bug làm cho việc nhật tu sai đi một chút. Vào lúc này, thông số đối với macro CONNECTION\_POINT\_ENTRY được kết sinh không có prefix D bắt buộc. Sau đây là connection map được sửa lại:

```
BEGIN_CONNECTION_POINT_MAP(CCoCar)
    // Ô hô ! wizard quên prefix D!
    // CONNECTION_POINT_ENTRY(IID_ICarEvents) // không tốt
    CONNECTION_POINT_ENTRY(DIID_ICarEvents) // đúng như vậy
END_CONNECTION_POINT_MAP()
```

(Thật ra, hình như bug này đã được sửa sai, vì khi chạy trên máy của người viết, không có bug này. Do đó, nếu máy của bạn có sai thì cho sửa lại như trên.)

Công cụ này còn thực hiện nhật tu thứ hai là thêm một lớp mới vào chuỗi kế thừa của lớp CoCar (**CProxy\_ICarEvents**). Lớp này thi công một hàm hành sự **Fire\_Exploded()** che dấu công việc triệu gọi danh sách các client hiện được kết nối. Lớp này nằm trên tập tin ClassicATLCOMServerCP.h).

Muốn phát động tình huống **Exploded**, bạn cho thêm hai biến thành viên mang tên **maxSpeed** (kiểu int) và **dead** (kiểu boolean), và chớ quên gán trị ban đầu trên hàm constructor **CoCar()** (trên CoCar.h). Biến **maxSpeed** tượng trưng cho tốc độ tối đa của COM car. Còn biến **dead** cho biết xe đã nổ tung chưa (dead = true khi xe bị nổ tung). Một khi các biến thành viên này đã được thêm vào header file, bạn cho nhật tu **SpeedUp()** như sau, rồi cho biên dịch:

```
STDMETHODIMP CCoCar::SpeedUp(int delta)
```



```

{ // Thêm delta cho curSpeed và kiểm tra tình trạng event
  curSpeed += delta;

  // Nếu ta hiện chưa chết thì vượt qua tốc độ tối đa
  if(curSpeed <= maxSpeed && !dead)
  { // Cho phát động tình huống và cho đặt để 'dead'
    CComBSTR msg("You are toast...");
    Fire_Exploded(msg.Detach());
    curSpeed = maxSpeed;
    dead = true;
  }
  return S_OK;
}

```

## 7.6.3 Cho tung ra một COM Error

Bạn nhớ cho là khi bạn tạo một ATL CoCar, bạn đưa thêm vào hỗ trợ Error Protocol. Đối với COM (hình 7-22, check box **ISupportErrorInfo** là On). Đối với những mục đích hiện hành, bạn không cần xem xét đến những chi tiết của các đối tượng COM error. Điều bạn cần biết là khi bạn muốn báo cáo một sai lầm khi sử dụng ATL Framework, bạn cho triệu gọi hàm kế thừa **Error()**. Để minh họa, bạn cho nhật tu phần thi công của **GetCurSpeed()** trả về một đối tượng COM error nếu người sử dụng muốn lấy tốc độ hiện hành của một chiếc xe đã chết máy:

```

STDMETHODIMP CCoCar::GetCurSpeed(int *currSp)
{
  // Gửi đi thông tin nếu xe chết máy
  if(!dead)
  {
    // trả về current speed
    *currSp = curSpeed;
    return S_OK;
  }
  else
  {
    *currSp = 0;
    Error("Sorry, this car has met it's maker");
    return E_FAIL;
  }
}

```

Tới đây coi như bạn đã tạo ra một COM class chịu hỗ trợ một single [default] interface. Ngoài ra, coclass này có thể phát ra thông tin COM error và những sai lầm đối với các khách hàng được kết nối. Một lần nữa bạn cho biên dịch lại dự án kiểm tra những sai lầm khi gõ vào. Nếu OK, ta đi tiếp.

## 7.6.4 Cho trưng ra một subobject nội bộ (và sử dụng SAFEARRAY)

Bạn sẽ thêm hai chức năng cuối cùng cho ATL COM Server. Chức năng đầu tiên là cho trưng ra một internal subobject từ kiểu dữ liệu CoCar mang tên **CoEngine**. Để bắt đầu, bạn thêm một **ATL Simple Object** mới bằng cách ra lệnh **Insert | New ATL Object**, cho đổi tên của [default] interface thành **IEngine**, và cho kiểu dữ liệu giao diện là [dual] trên Tab Attribute.

Kế tiếp, bạn sử dụng **Add Method Wizard** để thêm vào một hàm hành sự duy nhất đối với giao diện **IEngine**, mang tên **GetCylinders()**. Hàm này sẽ trả về một COM SAFEARRAY kiểu BSTR tượng trưng cho các tên cung (pet name) đối với mỗi cylinder được đặt vào mỗi máy xe. Sau đây là phần IDL:

```
// Inner engine interface
interface IEngine: IDispatch
{
    [id(1), helpstring("method GetCylinders")]
    HRESULT GetCylinders([out, retval] VARIANT* arCylinders);
};
```

Và đây là phần thi công đối với những máy chịu hỗ trợ 4 cylinders:

```
STDMETHODIMP CCoEngine::GetCylinders(VARIANT *arCylinders)
{
    // Khởi gán và đặt để kiểu dữ liệu của variant
    VariantInit(arCylinders);
    arCylinders->vt = VT_ARRAY | VT_BSTR; // một bản dãy chuỗi

    // Tạo một array.
    SAFEARRAY *pSA;
    SAFEARRAYBOUND bounds = {4, 0};
    pSA = SafeArrayCreate(VT_BSTR, 1, &bounds);

    // Cho điền array.
    BSTR *theStrings;
    SafeArrayAccessData(pSA, (void**)&theStrings);
    theStrings[0] = SysAllocString(L"Grinder");
    theStrings[1] = SysAllocString(L"Oily");
    theStrings[2] = SysAllocString(L"Thumper");
    theStrings[3] = SysAllocString(L"Crusher");
    SafeArrayUnaccessData(pSA);

    // Cho đặt để trị trả về
    arCylinders->parray = pSA;

    return S_OK;
}
```

Nếu trước đây bạn chưa hề làm việc với COM SAFEARRAY viết theo C++, có thể

bạn phải rừng mình trước đoạn mã kể trên. Ý kiến của SAFEARRAY là một bản dãy [oleautomation] kiểu dữ liệu tương thích tự mô tả được và duy trì giới hạn cao thấp của các mục tin nó chứa. Các kiểu dữ liệu SAFEARRAY sẽ được tạo ra, điền đầy dữ liệu và được thao tác bằng cách sử dụng các hàm thư viện COM. Một lần nữa, bạn khỏi đi vào chi tiết ở đây. Chỉ cần biết là tới đây hàm hành sự **GetCylinders()** cho phép ở ngoài nhận được một bản dãy kiểu dữ liệu BSTR.

Bước kế tiếp là cho phép COM client nhận được một qui chiếu giao diện hợp lệ **IEngine** từ kiểu dữ liệu **CoCar**. Muốn thế, bạn sử dụng COM containment chuẩn. **CoCar** hướng ngoại cung cấp việc truy cập **CoEngine** hướng nội sử dụng hàm hành sự **GetEngine()** được thêm vào giao diện **ICar**:

```
// bạn nhớ cho!!! Trả về một con trỏ giao diện đòi
// hỏi double indirection Do đó phải bảo đảm là định nghĩa của
// IEngine (nằm trong .idl) phải nằm trước định nghĩa của ICar, như
// vậy trình biên dịch MIDL có thể "thấy" định nghĩa giao diện.
// Nếu không sẽ báo sai. Rắc rối quá
interface ICar: IDispatch
{
    . . .

    [id(3), helpstring("method GetEngine")]
    HRESULT GetEngine([out, retval] IEngine** pEngine);
};
```

Việc thi công hàm **GetEngine()** dùng đến vài kiểu dữ liệu ATL, do đó nếu bạn không quen với những mục này, bạn chỉ cần biết là hàm hành sự static **CComObject<>::CreateInstance()** là hàm tạo ra **CoEngine**. Sau điểm này, ta đi truy vấn giao diện **IEngine** và trả nó về cho client, như sau:

```
STDMETHODIMP CCoCar::GetEngine(IEngine **pEngine)
{
    // Tạo một CoEngine rồi trả giao diện IEngine về cho client
    CComObject<CCoEngine> *pEng;
    CComObject<CCoEngine>::CreateInstance(&pEng);
    pEng->QueryInterface(IID_IEngine, (void**)pEngine);

    return S_OK;
}
```

Nếu bạn thật sự muốn truy cập những chi tiết sâu hơn của kiểu dữ liệu contained này, bạn có thể thêm noncreatable attribute vào định nghĩa IDL đối với **CoEngine**. Attribute này được sử dụng bởi vô số ngôn ngữ (trong ấy có Visual Basic 6.0) ngăn người sử dụng dùng **New** đối với item (nếu cô làm thì trình biên dịch sẽ phát ra sai lầm):

```
[
    uuid(32C07E17-F966-4EFD-B301-9729FE2D60B5),
```

```

        helpstring("CoEngine Class"), noncreatable
    ]
coclass CoEngine
{
    [default] interface IEngine;

};

```

Ngoài ra, bạn phải cho nhật tu ATL OBJECT\_MAP (trên ClassicATLCOMServer.cpp) cho đánh dấu CoEngine là kiểu dữ liệu noncreatable, ngăn việc tạo trực tiếp bởi một clien nằm ngoài, như sau:

```

BEGIN_OBJECT_MAP(ObjectMap)
    OBJECT_ENTRY(CLSID_CoCar, CCoCar)
    OBJECT_ENTRY_NON_CREATEABLE(CCoEngine)
END_OBJECT_MAP()

```

Cuối cùng trên tập tin CoCar.cpp, bạn phải thêm chỉ thị #include “CoEngine.h”, trước khi cho biên dịch lại.

## 7.6.5 Bước cuối cùng: cấu hình hóa một IDL Enumeration

Đây là bước cuối cùng trước khi bạn có thể tạo một proxy đối với COM server này. Bạn cho mở tập tin IDL của dự án (trên ClassicATLCOMServer.idl) và định nghĩa một COM enumeration duy nhất mang tên **CarType** trực tiếp ngay dưới câu lệnh import ở đầu tập tin như sau:

```

// enum này dùng nhận diện kiểu xe.
typedef enum CarType {Jetta, BMW, Ford, Colt} CarType;

```

Muốn cho thế giới bên ngoài nhận được enum **CarType** bạn phải thêm một hàm hành sự đối với giao diện ICar, mang tên **GetCarType()** như sau:

```

interface ICar: IDispatch
{
    [id(1), helpstring("method SpeedUp")]
    HRESULT SpeedUp([in] int delta);

    [id(2), helpstring("method GetCurSpeed")]
    HRESULT GetCurSpeed([out, retval] int* currSp);

    [id(3), helpstring("method GetEngine")]
    HRESULT GetEngine([out, retval] IEngine** pEngine);

    [id(4), helpstring("method GetCarType")]
    HRESULT GetCarType([out, retval] CarType* ct);
};

```

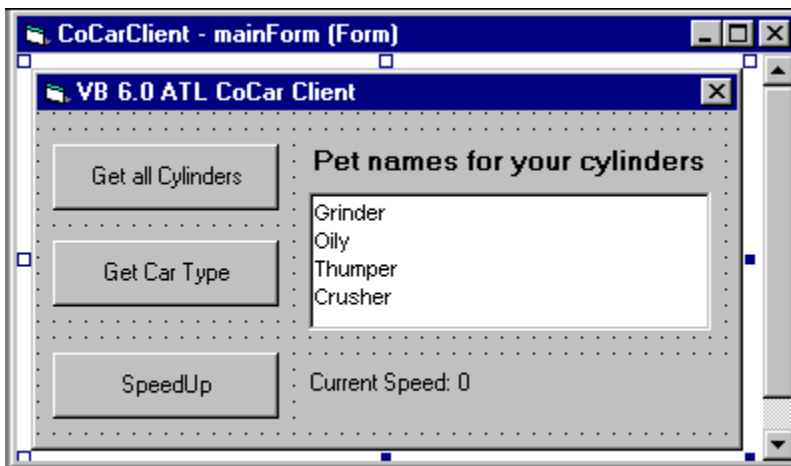
Việc thi công hàm hành sự **GetCarType()** sẽ tạo và đặt đề trị của enum [out, retval] **CarType** để cho COM client dùng, như sau (trong CoCar.cpp):

```
STDMETHODIMP CCoCar::GetCarType(CarType *ct)
{
    *ct = Colt;    // chọn xe ưa thích của bạn
    return S_OK;
}
```

Chúng tôi tin chắc là bạn không muốn học ATL trong một quyển sách nói về C#. <sup>22</sup> Nhưng chương này lại nói tất cả về COM và .NET interoperability. Lý do bạn phải mất thời giờ để xây dựng một server phức tạp là vì lập trình ATL khá phức tạp. Ở đây bạn có một COM server chịu hỗ trợ các đối tượng Error, connection point, COM containment, enumeration và vài kiểu dữ liệu COM cốt lõi chẳng hạn SAFEARRAY và BSTR.

Bây giờ bạn đã có một ATL COM server được tạo một cách trọn vẹn, bạn có thể tạo một proxy và kiểm tra xem các kiểu dữ liệu COM được ánh xạ lên .NET thế nào. Trước khi thực hiện điều này, ta thử sử dụng CoCar trên một COM client viết theo Visual Basic 6.0.

## 7.6.6 Thử tạo một Visual Basic 6.0 Test Client



Hình 7-25: VB 6.0 COM Client

Trước khi bạn tạo một managed client, bạn thử tạo một COM client cổ điển sử dụng Visual Basic 6.0. Chúng tôi chỉ khảo sát những điểm chính yếu. Hình 7-25 cho thấy biểu mẫu ứng dụng client.

Khi Form được nạp, bạn tạo một thể hiện của Car

type (được khai báo sử dụng từ chốt WithEvents đáp ứng tình huống Exploded). Hàm thủ lý tình huống **Exploded** đơn giản cho hiển thị một thông điệp gọi đi bởi chiếc xe chết tiệt, như sau:

<sup>22</sup> Muốn tìm hiểu sâu ATL, bạn có thể tìm đọc sách “Lập trình Windows với Visual C++, tập 5, của Dương Quang Thiện.

```
Private Sub myCar_Exploded(ByVal deadMsg As String)
    MsgBox deadMsg, , "Message from CoCar!"
End Sub
```

Nút <SpeedUp> làm việc này. Bạn còn nhớ khi bạn tăng tốc độ lên tối đa, tình huống Exploded (nổ tung) sẽ xảy ra phát đi bởi CoCar. Ngoài ra, khi người sử dụng muốn tăng tốc độ đối với một chiếc xe đã bị nổ tung thì một đối tượng COM Error được gọi đi (được tiếp nhận bởi Visual Basic sử dụng On Error Goto), như sau đây:

```
Private Sub btnSpeedUp_Click()
    On Error GoTo OOPS

    myCar.SpeedUp 50
    Label2.Caption = "Current Speed: " & myCar.GetCurSpeed

Exit Sub

OOPS:
    MsgBox Err.Description, , "Error from car!"
    Resume Next
End Sub
```

Nút <Get all Cylinders> sẽ nhận về giao diện **IEngine** từ **CoCar**, rồi triệu gọi hàm hành sự **GetCylinders()** như sau:

```
Private Sub btnGetCylinders_Click()

    lstCylinderList.Clear

    ' Trước tiên cần lấy máy
    Dim q As CoEngine
    Set q = myCar.GetEngine

    ' Bây giờ đi lấy cylinders.
    Dim strs As Variant
    strs = q.GetCylinders

    ' Lấy mỗi tên từ SAFEARRAY.
    Dim upper As Integer
    Dim i As Integer
    upper = UBound(strs)
    For i = 0 To upper
        lstCylinderList.AddItem strs(i)
    Next i

End Sub
```

Cuối cùng, nút <Get Car Type> đi lấy các kiểu xe. Đoạn mã như sau:

```

Private Sub btnGetCarType_Click()

    Dim i As CarType
    i = myCar.GetCarType

    Dim enumVals(4) As String
    enumVals(0) = "Jetta"
    enumVals(1) = "BMW"
    enumVals(2) = "Ford"
    enumVals(3) = "Colt"

    MsgBox enumVals(i), , "Car Type:"

End Sub

```

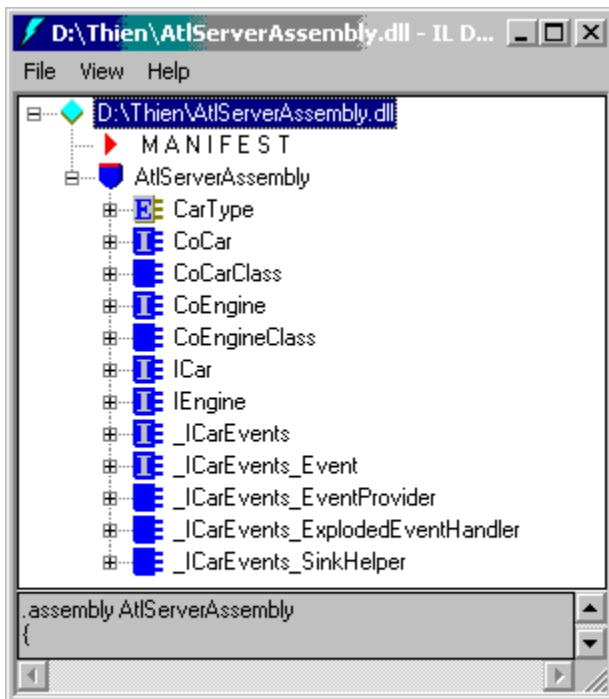
Biểu mẫu được nạp thông qua đoạn mã như sau:

```

Private Sub Form_Load()
    Set myCar = New CoCar
End Sub

```

## 7.6.7 Xây dựng Assembly và xem xét tiến trình chuyển đổi



Hình 7-26: Managed ATL CoCar

Sau khi bạn đã tạo ra **ClassicATLCOMServer.dll** chạy tốt, bạn có thể chép qua môi trường .NET. Bạn cho mở cửa sổ command prompt và cho trình tiện ích **tlbimp.exe** để tạo một assembly mới tượng trưng cho **ClassicATLCOMServer.dll** như sau:

```

tlbimp
classicatlcomserver.dll
/out:AtlServerAssembly.dll

```

Bây giờ, bạn cho nạp assembly này vào **ILDasm.exe** và kiểm tra các kiểu dữ liệu được kết sinh (hình 7-26). Như bạn có thể thấy, các kiểu dữ liệu COM **CarType**, **CoCar** và **CoEngine** được ánh xạ lên các lớp .NET tương đương. Các hàm hỗ trợ tình huống cũng sẽ

được xét đến sau. Trong lúc này, chúng tôi sẽ bàn đến chi tiết việc chuyển đổi này, bắt đầu từ COM type library.

### 7.6.7.1 Type Library Conversion

Như bạn có thể thấy, mỗi kiểu dữ liệu COM (enum, coclass hoặc customer interface) được liệt kê trên câu lệnh thư viện COM server sẽ được dùng để điền namespace .NET được kết sinh.

Một điểm bổ sung đáng chú ý là attribute [version] của type library sẽ được dùng chỉ định phiên bản của assembly. Như vậy, nếu bạn nhậ tu phiên bản IDL attribute sẽ như sau:

```
// ClassicATLCOMServer.idl: IDL source for ClassicATLCOMServer.dll
//

// This file will be processed by the MIDL tool to produce
// the type library (ClassicATLCOMServer.tlb) and marshalling code.
. . .
[
    uuid(BCDCA5AD-4CB8-11D8-A6BC-0000C0A45093), version(9.7),
    helpstring("ClassicATLCOMServer 1.0 Type Library")
]
library CLASSICATLCOMSERVERLib
{
    . . .
};
```

mà bạn có thể thấy trong bảng liệt in sau đây trên manifest:

```
.assembly AtlServerAssembly
{
    . . .
    .hash algorithm 0x00008004
    .ver 9:7:0:0
}
```

### 7.6.7.2 COM Interface Conversion

Khi một giao diện COM được tượng trưng như là một kiểu dữ liệu .NET, nó được phép sử dụng những attribute khác nhau lấy từ namespace **System.Runtime.InteropServices**. Trước tiên, là attribute **GuidAttribute**, dùng trong trường hợp này sưu liệu mã nhận diện IID của giao diện, như được khai báo bởi [uuid] trên tập tin IDL.

Tiếp theo là kiểu dữ liệu **InterfaceTypeAttribute** dùng cho biết giao diện được định nghĩa thế nào lúc ban đầu theo cú pháp IDL (custom, dual, hoặc dispinterface). Attribute này có thể được gán bất cứ trị nào từ enumeration **ComInterfaceType**. Xem bảng 7-06.



**Bảng 7-06: Enumeration COMInterfaceType**

Thành viên	Mô tả
<b>InterfacesDual</b>	Cho biết giao diện phải được trung ra cho COM như là giao diện dual (hai mặt).
<b>InterfacesDispatch</b>	Cho biết giao diện phải được trung ra như là một dispinterface.
<b>InterfacesUnknown</b>	Cho biết giao diện phải được trung ra như là một giao diện được dẫn xuất từ IUnknown, đối nghịch với dispinterface hoặc dual interface.

Điều kỳ lạ là nếu proxy tượng trưng cho một dual interface (như trong trường hợp này). thì attribute này bị bỏ qua. Thay vào đó, dual interface được ghi nhận sử dụng attribute **TypeLibTypeAttribute**, thường được dùng suu liệu những khía cạnh khác nhau của [dual] interface.

### 7.6.7.3 Parameter Attribute Conversion

Trong COM cổ điển, các thông số thường được cấu hình sử dụng một tập hợp attribute IDL. Những attribute này được dùng cho biết hướng đi (nhập, xuất) của một đối mục nào đó và để ấn định việc quản lý ký ức một cách thích ứng. Các coclass ATL sẽ sử dụng các attribute IDL **[in]**, và **[out, retval]**. Ngoài ra, một thông số của hàm hành sự COM có thể được đánh dấu bởi attribute **[out]** hoặc **[in, out]**. Để minh họa mọi khả năng, giả sử bạn định nghĩa giao diện **IPParams** trên IDL như sau:

```
interface IPParams: IDispatch
{
    // [in] params được cấp phát bởi phía triệu gọi và
    // chuyển cho hàm hành sự
    [id(1)] HRESULT OnlyInParams([in] int x, [in] int y);

    // [out] params được điền bởi phía bị triệu gọi
    [id(2)] HRESULT OnlyOutParams([out] int* x, [out] int* y);

    // [retval] type ánh xạ một thông số xuất về trị trả về vật lý
    // của hàm hành sự (thí dụ, Visual Basic có thể triệu gọi
    // như sau "ans = Retval()")
    [id(3)] HRESULT RetVal([out, retval] int* answer);

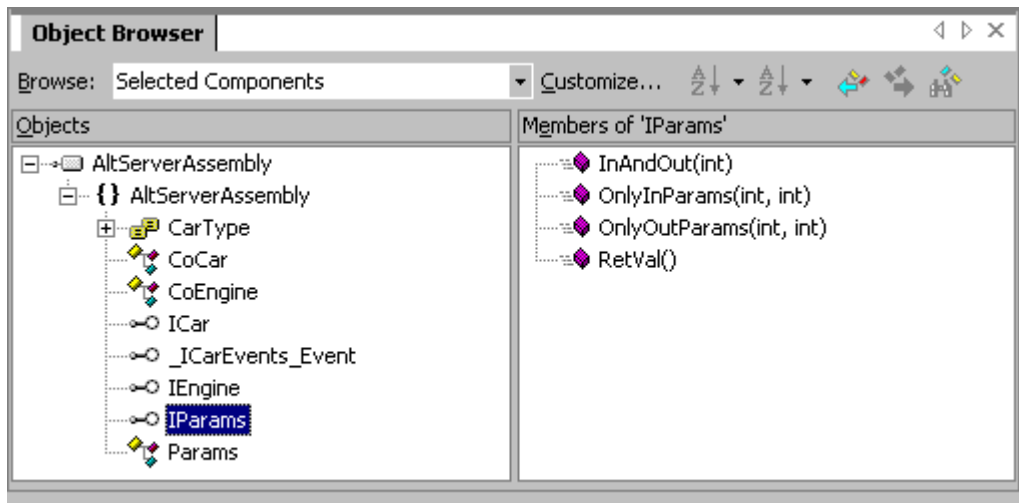
    // [in, out] param được chuyển cho phía bị triệu gọi
    // và có thể bị thay đổi trong tiến trình triệu gọi hàm
    [id(4)] HRESULT InAndOut([in, out] int* byRefParam);
};
```

Một khi bạn cho chạy định nghĩa giao diện này thông qua trình tiện ích **tlbimp.exe**, bạn sẽ thấy là những mục này sẽ được ánh xạ lên những từ chốt quen thuộc **out** và **ref** của C#, như theo hình 7-27 được hiển thị dùng đến Visual Studio .NET Object Browser:

Bảng 7-07 sau đây cho thấy việc chuyển đổi các thông số từ IDL qua C#:

**Bảng 7-07: Ánh xạ các IDL Parameter Attributes lên các từ chốt C#**

IDL Parameter Attribute	C# Parameter Keyword	Mô tả
[in]	Không có từ chốt. Đây là chiều hướng giả định phải đi	Hàm được triệu gọi sẽ nhận một bản sao dữ liệu
[out]	out	Trị được gán trong lòng hàm bị triệu gọi, và sẽ được trả về cho phía triệu gọi.
[in, out]	ref	Trị được gán bởi phía triệu gọi, nhưng có thể được cấp phát lại bởi hàm bị triệu gọi.
[out, retval]	not announced	Các kiểu dữ liệu này trở thành trị trả về vật lý của hàm. Nếu [out, retval] không được định nghĩa một cách hình thức, thì nó sẽ được chuyển đổi thành một kiểu dữ liệu trả về void.



**Hình 7-27: Managed IPParams interface**

#### 7.6.7.4 Interface Hierarchy Conversion

Mặc dù, bạn không định nghĩa một đẳng cấp đối với COM Interface trong dự án ATL, giả sử bạn quyết định dẫn xuất một giao diện mới từ **IEngine**, cho mang tên **ITurboEngine**. Sau đây là phần định nghĩa IDL:

```
interface ITurboEngine: IEngine
{
    HRESULT PowerBoost();
};
```

Dựa theo tiến trình chuyển đổi, một giao diện được dẫn xuất thường được tượng trưng như là một union của tất cả các hàm hành sự được định nghĩa trong một chuỗi kế thừa. Do đó, nếu bạn dùng trình tiện ích ILDasm.exe quan sát, bạn sẽ thấy là **ITurboEngine** không những hỗ trợ **PowerBoost()** mà còn hỗ trợ **GetCylinders()**, hàm sau này thuộc giao diện **IEngine**. Ngoài ra, bạn sẽ thấy dòng chữ:

```
implements AtlServerAssembly.IEngine
```

trên assembly nghĩa là tag [implement] cho biết tên của giao diện cơ sở.

### 7.6.7.5 Coclass (và COM Properties) Conversion

Như đã được minh họa bởi thí dụ Visual Basic COM Server đi trước, khi assembly được kết sinh bởi trình tiện ích **tlbimp.exe**, bạn sẽ nhận được những kiểu dữ liệu .NET đối với mỗi giao diện riêng rẽ kể cả những coclass. Do đó, bạn có thể sử dụng một kiểu dữ liệu CoCar mới theo hai cách tiếp cận khác nhau. Trước tiên, bạn có thể tạo một thể hiện trực tiếp của coclass cho phép truy cập mỗi thành viên giao diện, như theo thí dụ sau đây:

```
// Ở đây, bạn có thể thật sự làm việc với
// [default] ICar interface
CoCar viper = new CoCar();
viper.SpeedUp(30);
```

Một cách khác, bạn có thể yêu cầu một cách tường minh đối với [default] interface như sau:

```
// Đi lấy ICar một cách tường minh
CoCar viper = new CoCar();
ICar ic = (ICar) viper;
ic.SpeedUp(30);
```

Trong thí dụ ATL hiện hành, các kiểu dữ liệu CoCar và CoEngine mỗi lớp cho thi công một [default] interface đơn lẻ. Tuy nhiên, giả sử bạn định nghĩa một giao diện bổ sung **IDriverInfo** như sau, chịu hỗ trợ một thuộc tính **DriverName** đơn độc (kiểu BSTR);

```
interface IDriverInfo:IDispatch
{
    [ propget, id(1),helpstring("property DriverName")]
    HRESULT DriverName([out, retval] BSTR *pVal);
```

```
[ propput, id(1),helpstring("property DriverName")]
HRESULT DriverName([in] BSTR newVal);

};
```

Ta cũng giả sử thêm là CoCar bây giờ thi công giao diện mới này như sau:

```
coclass CoCar
{ [default] interface ICar;
  interface IDriverInfo;
  [default, source] dispinterface _ICarEvents;
};
```

Bây giờ, ta cho rằng **CoCar** hỗ trợ 2 custom interface, có thể bạn muốn biết giờ đây việc này được tượng trưng thế nào theo managed code. Như bạn có thể đoán ra, managed code sẽ hỗ trợ mỗi thành viên được định nghĩa bởi các giao diện được hỗ trợ. Nói cách khác, nếu giờ đây bạn tạo một thể hiện của **CoCar**, bạn có thể triệu gọi hàm **SpeedUp()**, **GetCurSpeed()**, **GetEngine()** và **GetCarType()** cũng như thao tác lên thuộc tính **DriverName**, như sau:

```
// Bạn để ý ta có thể truy cập thuộc tính được định nghĩa bởi
// IDriverInfo trực tiếp từ coclass chịu hỗ trợ IDriverInfo
CoCar viper = new CoCar();
viper.DriverName = "Fred";
Console.WriteLine(viper.DriverName);
```

Nếu muốn, bạn cũng có thể truy cập một cách tường minh giao diện **IDriverInfo** như sau:

```
// Get và Set thuộc tính sử dụng giao diện IDriverInfo
CoCar viper = new CoCar();
IDriverInfo idi = (IDriverInfo)viper;
idi.DriverName = "Fred";
Console.WriteLine("Name of driver is: " + idi.DriverName);
```

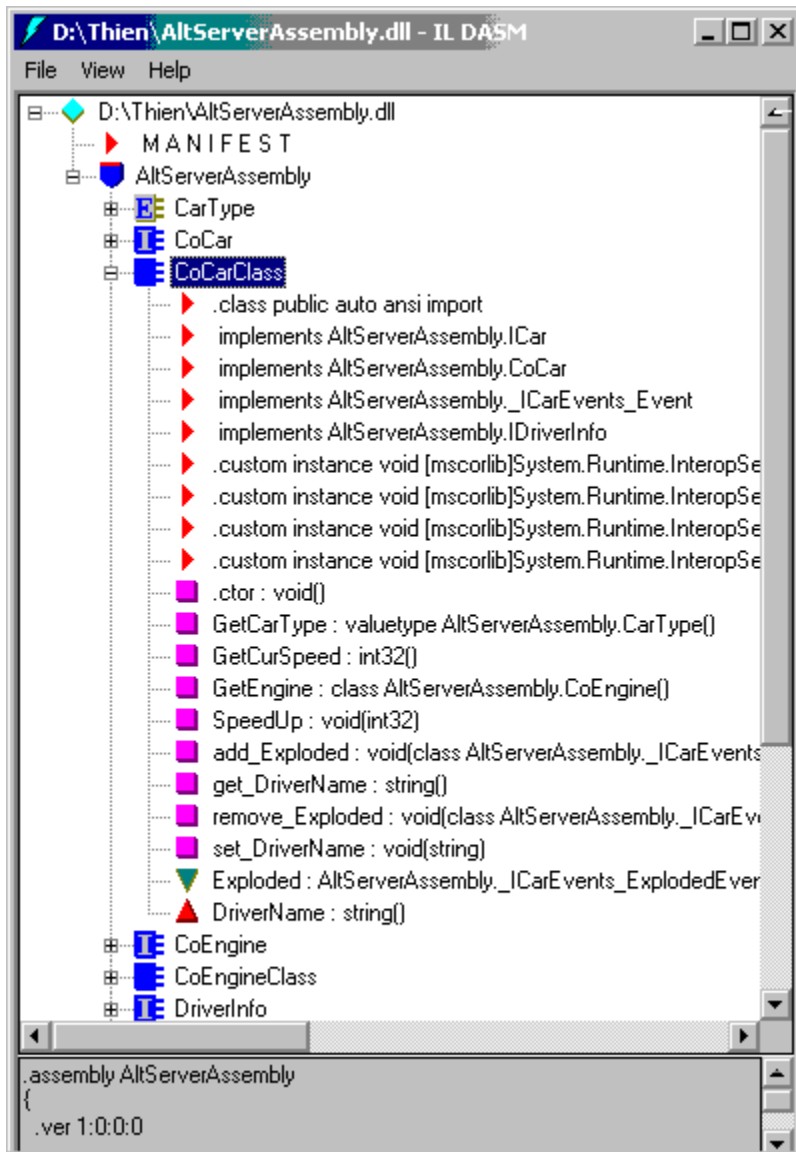
Sau khi bạn đã tạo xong dự án ClassicATLCOMServer xong trên Visual C++, bạn cho chép cả thư mục dự án này qua môi trường Visual Studio .NET. Chẳng hạn qua D:\Thien. Tiếp theo bạn cho đăng ký sử dụng trình tiện ích **regsvr32.exe** trên command line như sau:

```
regsvr32 d:\thien\ClassicATLCOMServer\debug\ClassicATLCOMServer.dll
```

Khi đăng ký thành công, bạn dùng trình tiện ích **tlbimp.exe** để nhập khẩu **ClassicATLCOMServer.dll** vào assembly .NET như sau:

```
tlbimp ClassicATLCOMServer\debug\ClassicATLCOMServer.dll
/out:atlserverassembly.dll
```

Khi chuyển đổi thành công từ COM qua .NET, bạn sử dụng trình tiện ích **ILDasm.exe** để khảo sát assembly **atlsverassembly.dll**.



**Hình 7-28: Thêm IDriverInfo**

Khá lý kỳ, là nếu bạn kiểm tra kiểu dữ liệu **CoCar** sử dụng **ILDasm.exe**, bạn sẽ ngạc nhiên là kiểu dữ liệu **CoCar** không trực tiếp định nghĩa các thành viên này. Tuy nhiên, trong trường hợp này, nếu bạn xem kỹ bạn sẽ thấy **CoCar** được dẫn xuất từ một kiểu dữ

liệu được kết sinh **CoCarClass**, và lớp trung gian này định nghĩa các thành viên của các giao diện **ICar** và **IDriverInfo**. Xem hình 7-28 trang trước.

Khi bạn kiểm tra type metadata, bạn sẽ thấy dẫn xuất sau đây của **CoCarClass**:

```
.class public auto ansi import CoCarClass
    extends [mscorlib]System.Object
    implements AltServerAssembly.ICar,
        AltServerAssembly.CoCar,
        AltServerAssembly.ICarEvents_Event,
        AltServerAssembly.IDriverInfo
{
    // TypeLibTypeAttribute attribute...
    // ComSourceInterfaceAttribute attribute...
    // GuidAttribute attribute...
    // ClassInterfaceAttribute attribute...
} // end of class _CoCar
```

### 7.6.7.6 COM Enumeration Conversion

COM enumeration được ánh xạ lên các kiểu dữ liệu managed từ **System.Enum**. Do đó, bạn có thể sử dụng bất cứ các thành viên nào được hỗ trợ. Sau đây là một thí dụ. Bạn tạo một dự án C# Console Application, cho mang tên **TestATLServer**. Bạn nhớ sử dụng các chỉ thị using như sau:

```
using System.Runtime.InteropServices;
using atlcomserverassembly;
```

và dùng khung đối thoại **Add Reference** qui chiếu về tập tin **AltComServerAssembly.dll**.

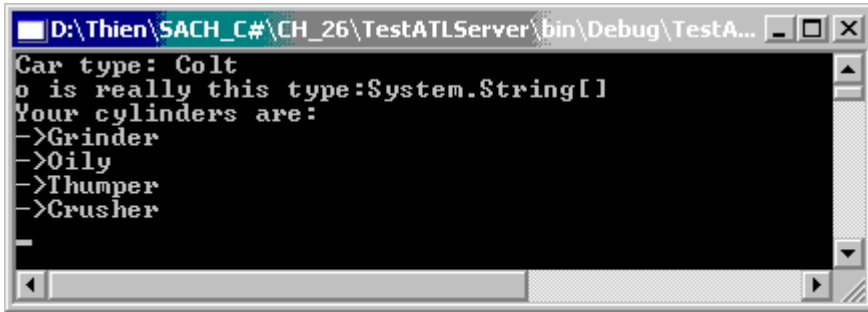
Xong rồi bạn thêm đoạn mã vào hàm hành sự **Main()**:

```
public static int Main(string[] args)
{
    // bắt đầu tạo một đối tượng Car
    CoCar viper = new CoCar();

    // đi lấy loại xe
    CarType t = viper.GetCarType();
    Console.WriteLine("Car type: ", t.ToString());

    return 0;
}
```

Kết quả là hình 7-29, hàng đầu tiên.



Hình 7-29: Kết xuất việc rảo qua SAFEARRAY

### 7.6.7.7 Làm việc với COM SAFEARRAY

Tiếp theo, ta thử xem COM SAFEARRAY được tượng trưng như thế nào ở managed code. Như bạn đã biết, giao diện **IEngine** hỗ trợ đơn độc một hàm hành mang tên **GetCylinders()**, trả về một bản dãy kiểu BSTR. Client nằm ngoài truy cập giao diện **IEngine** bằng cách triệu gọi hàm **GetEngine()** đối với một đối tượng **CoCar** hiện hữu. Như vậy, bạn có thể bắt đầu viết như sau:

```
CoCar viper = new CoCar();           // tạo một thể hiện CoCar
IEngine e = viper.GetEngine();        // lấy qui chiếu giao diện
IEngine
object o = e.GetCylinders();          // yêu cầu engine cung cấp SAFEARRAY
```

Bạn nhớ lại SAFEARRAY được khai báo như là một bản dãy kiểu dữ liệu VARIANT, sử dụng type flags **VT\_ARRAY** và **VT\_BSTR**, như sau:

```
STDMETHODIMP CCoEngine::GetCylinders(VARIANT *arCylinders)
{ // cho khởi gán và đặt để kiểu dữ liệu variant
  VariantInit(arCylinders);
  arCylinders->vt = VT_ARRAY | VT_BSTR;

  // Cho gỡ bỏ COM quá phức tạp để bạn khỏi điên lên

  // Cho đặt để trị trả về
  arCylinders->parray = pSA;
  return S_OK;
}
```

Theo cái nhìn của .NET, một kiểu dữ liệu VARIANT được tượng trưng bởi một thể hiện của **System.Object**. Tuy nhiên, vì bạn đặt để vùng mục tin **vt** là VARIANT sử dụng type flags **VT\_ARRAY** và **VT\_BSTR**, bạn sẽ thấy **System.String[]** được in ra trên console:

```
// 'o' nghĩ anh ta là gì thế?
```

```

IEngine e = viper.GetEngine();
object o = e.GetCylinders();           // yêu cầu engine cung cấp SAFEARRAY

// o thuộc kiểu dữ liệu System.String[]
Console.WriteLine("o is really this type: {0}", o);

```

Muốn in ra petname đối với mỗi cylinder, bạn có thể viết ra như sau:

```

String[] cylinders = (string[])o;      // lấy bản dãy các chuỗi
Console.WriteLine("Your cylinders are: ");
foreach(string s in cylinders)
{   Console.WriteLine(">" + s);
}

```

Kết xuất như sau, hình 7-29, từ hàng thứ hai trở đi.

Điểm kế tiếp là kiểm tra tiến trình móc nối vào tình huống **Exploded** của CoCar.

### 7.6.7.8 Chặn hứng các tình huống COM

Trong chương 12, tập I, “Ủy quyền và tình huống”, bạn đã biết qua mô hình tình huống .NET. Bạn nhớ lại kiến trúc này dựa trên việc ủy thác flow of logic từ một phần ứng dụng này qua một phần khác Đối tượng lo việc chuyển đi một yêu cầu là một kiểu dữ liệu được dẫn xuất từ **System.MulticastDelegate**. Client có thể thêm một đích hoặc gỡ bỏ một đích khỏi danh sách nội tại sử dụng các tác tử nạp chồng += và -=.

Khi trình tiện ích **tlbimp.exe** gặp phải một [source] interface trên type library của COM server, thì nó đáp ứng bằng cách tạo ra một số kiểu dữ liệu managed cho bao trọn kiến trúc cấp thấp của COM connection point. Sử dụng các kiểu dữ liệu này, bạn coi như là thêm một thành viên vào danh sách nội tại kết nối của **System.MulticastDelegate**. Lẽ dĩ nhiên, nằm sau hậu trường là một proxy ánh xạ tình huống COM đi vào qua đối tượng tương đương phía managed. Bạn nhớ cho **CoCar** unmanaged được định nghĩa như sau trên outbound interface:

```

dispinterface _ICarEvents
{
    properties:
    methods:
        [id(1), helpstring("method Exploded")
        HRESULT Exploded([in] BSTR deadMsg);
};

```

Trình tiện ích **tlbimp.exe** sẽ đáp ứng bằng cách tạo ra một tập hợp các kiểu dữ liệu được dùng ánh xạ kiến trúc COM connection point lên hệ thống .NET delegate event. Bảng 7-08 mô tả các kiểu dữ liệu này:



**Bảng 7-08: COM Event Helper Types**

Các kiểu dữ liệu được kết sinh (dựa trên <code>_ICarEvents [source] interface</code> )	Mô tả
<code>_ICarEvents</code>	Đây là định nghĩa managed đối với outbound interface (và thường không được sử dụng trực tiếp).
<code>_ICarEvents_Event</code>	Đây là một managed interface định nghĩa các thành viên thêm vào và bỏ ra được dùng để thêm vào (hoặc gỡ bỏ) một hàm hành sự vào danh sách kết nối của <b>System.MulticastDelegate</b> . Kiểu dữ liệu này thường cũng không được dùng trực tiếp.
<code>_ICarEvents_ExplodedEventHandler</code>	Đây là managed delegate (được dẫn xuất từ <b>System.MulticastDelegate</b> ). Kiểu dữ liệu trả về đối với một hàm thụ lý tình huống managed nào đó phải trả về một số nguyên. Các thông số sẽ ánh xạ lên COM event nguyên thủy.
<code>_ICarEvents_SinkHelper</code>	Lớp được kết sinh này cho thi công outbound interface trên một đối tượng sink ăn ý với .NET. Lớp này gán cookie <sup>23</sup> được kết sinh bởi kiểu dữ liệu COM cho biến thành viên <code>m_dwCookie</code> . Ngoài ra, lớp này duy trì một biến thành viên nội tại ( <code>m_ExplodedDelegate</code> ) tượng trưng cho outbound interface ( <code>_ICarEvents_ExplodedEvent Handler</code> ).

Ngoài những kiểu dữ liệu được kết sinh kể trên, coclass (`CoCarClass`) định nghĩa outbound interface cũng được nhậ tu luôn. Trước tiên, (và quan trọng nhất) bạn có một tình huống mới mang tên `Exploded` như sau:

```
.class public auto ansi import CoCarClass
    extends [mscorlib]System.Object
    implements AtlServerAssembly.ICar,
        AtlServerAssembly.CoCar,
        AtlServerAssembly._ICarEvents_Event,
        AtlServerAssembly.IDriverInfo
    {
        . . .
    } // end of class CoCarClass
```

`CoCarClass` managed cũng sẽ có hai thành viên private duy trì kết nối với COM event source (`add_X` và `remove_X`), chẳng hạn **`add_Exploded`** và **`remove_Exploded`**. Xem hình 7-28.

Nếu bạn cho kiểm tra đoạn mã IL đối với **`add_Exploded()`** bạn sẽ thấy một kiểu dữ liệu **`_ICarEvents_SinkHelper`** được tạo giúp bạn. Sau điểm này, assembly proxy sẽ

<sup>23</sup> Cookie, không dịch được, ám chỉ một mẫu thông tin được trữ trên một browser, thường gồm những cặp `name = value`.

nhận được đúng giao diện `IConnectionPoint` từ ATL `CoCar`, triệu gọi hàm **Advise()**, và cache cookie được trả về.

### *Móc nối với tình huống COM*

Bây giờ bạn đã biết các tình huống COM được thụ lý thế nào rồi, bạn có thể dễ dàng học cách móc nối với tình huống COM từ managed code. Như bạn có thể thấy, nó cũng giống như với tiến trình làm việc với delegate của .NET.

```
public class CoCarClient
{
    // Hàm này sẽ được triệu gọi khi đối tượng Car gọi đi tình huống
    // Delegation target phải trả về một int
    public static int ExplodedHandler(String msg)
    {
        Console.WriteLine("\nCar says: (COM Events)\n->" + msg + "\n");
        return 0;
    }

    public static int Main(string[] args)
    {
        CoCar viper = new CoCar();
        viper.Exploded += new _ICarEvents_ExplodedEventHandler(
                                                                    ExplodedHandler);

        // Làm gì đó kích hoạt tình huống
        for(int i = 0; i < 5; i++)
        {
            try
            {
                viper.SpeedUp(50);
                Console.WriteLine("->Curr speed is: "
                                + viper.GetCurSpeed());
            }
            catch(Exception ex)
            {
                Console.WriteLine("->COM error! " + ex.Message + "\n");
            }
        }
    }
}
```

Bạn chỉ cần thiết lập một delegate với tình huống Exploded như sau:

```
viper.Exploded += new _ICarEvents_ExplodedEventHandler(
                                                                    ExplodedHandler);
```

Khi `CoCar` phát pháo tình huống Exploded (khi tốc độ trên giới hạn tối đa) hàm thụ lý tình huống (ExplodedHandler) sẽ tự động được triệu gọi.

### *7.6.7.9 Thụ lý COM Error*

Bạn để ý bạn cho bao trọn logic hàm Speedup() trong một khối try/catch. Nếu người sử dụng cố tăng tốc độ của một chiếc xe đã bị nổ tung trước đó, thì coclass sẽ trả về một đối tượng COM error, được ánh xạ lên biệt lệ .NET.

## 7.6.8 Toàn bộ ứng dụng C# Client

Tới đây coi như bạn đã hiểu sâu từng thành phần COM được biểu diễn theo .NET. Sau đây là toàn bộ ứng dụng C# client sử dụng proxy được kết sinh:

### *Thí dụ 7-10: Ứng dụng CoCarClient viết theo C#*

\*\*\*\*\*

```
using System;
using AtlServerAssembly;
using System.Reflection;

namespace CoCarClient
{
    public class CoCarClient
    {
        public static int ExplodedHandler(String msg)
        {
            Console.WriteLine("\nCar says: (COM Events)\n->" + msg
                               + "\n");
            return 0;
        }

        public static int Main(string[] args)
        {
            // Tạo một đối tượng Car
            CoCar viper = new CoCar();

            // Thụ lý tình huống Exploded
            viper.Exploded += new _ICarEvents_ExplodedEventHandler(
                               ExplodedHandler);

            // Set (và Get) tên lái xe
            viper.DriverName = "Fred";
            Console.WriteLine("Driver is named: (COM property)\n->"
                               + viper.DriverName + "\n");

            // liệt kê kiểu xe
            CarType t = viper.GetCarType();
            Console.WriteLine("Car type is: (COM enum)\n->"
                               + t.ToString() + "\n");

            // Đi lấy engine & cylinder
            IEngine e = viper.GetEngine();
            object o = e.GetCylinders();

            // Đi lấy bản dãy chuỗi và in mỗi mục tin
            String[] cylinders = (string[])o;
            Console.WriteLine("Your cylinders are: ");
            foreach(string s in cylinders)
            {
                Console.WriteLine("->" + s);
            }
        }
    }
}
```

```
// Bây giờ tăng tốc độ để kích hoạt tình huống
for(int i = 0; i < 5; i++)
{
    try
    {
        viper.SpeedUp(50);
        Console.WriteLine("->Curr speed is: "_par
                           + viper.GetCurSpeed());
    }
    catch(Exception ex)
    {
        Console.WriteLine("->COM error! " + ex.Message + "\n");
    }
}
Console.ReadLine();
return 0;
}
}
}
*****
```

Bạn thử cho chạy xem kết quả ra sao. Có đúng ý bạn muốn không?

## 7.7 Tìm hiểu Interoperability từ COM chuyển qua .NET

Đề mục kế tiếp mà chúng tôi muốn bàn đến là kịch bản interoperability ngược lại mà chúng ta vừa xem qua: một COM class triệu gọi một kiểu dữ liệu .NET. Bạn có thể đoán ra là trường hợp này sẽ ít xảy ra, nhưng cũng đáng cho ta tìm hiểu.

Muốn một COM coclass sử dụng một kiểu dữ liệu .NET, bạn cần “lừa” coclass làm thế nào cho nó có cảm tưởng kiểu dữ liệu managed thật ra là một kiểu dữ liệu unmanaged. Trong thực chất, bạn cần cho phép coclass tương tác với kiểu dữ liệu .NET sử dụng đến chức năng cung cấp bởi kiến trúc COM. Thí dụ, kiểu dữ liệu COM phải có khả năng nhận được những giao diện mới thông qua việc triệu gọi hàm **QueryInterface()**, mô phỏng việc quản lý ký ức sử dụng hàm hành sự **AddRef()** và **Release()**, sử dụng nghi thức **COM Connection Point**, v.v..

Ngoài việc lừa COM client, COM to .NET interoperability còn lừa COM runtime. Như bạn đã biết, một COM Server cổ điển được khởi động sử dụng SCM (Service Control Manager). Muốn việc này xảy ra, SCM phải dò tìm vô số thông tin trong system registry (ProgID, CLSID, IID v.v..). Vấn đề ở đây là các assembly .NET không có đăng ký gì cả.

Về mặt cốt lõi, muốn cho các assembly .NET xài được đối với COM client cổ điển, bạn phải chuẩn bị các bước sau đây:

- Cho đăng ký assembly .NET lên system registry như vậy SCM mới có thể tìm ra nơi tá túc của assembly. Trong bước này bạn sẽ sử dụng trình tiện ích **Regasm.exe** (tắt chữ Register Assembly).
- Cho kết sinh một COM type library (\*.tlb), dựa trên .NET metadata, như vậy COM client cổ điển mới có thể tương tác với các kiểu dữ liệu được trưng ra. Trong bước này bạn sẽ dùng trình tiện ích **TlbExp.exe** (tắt chữ Type Library Export).
- Cho triển khai assembly trên cùng thư mục với COM client hoặc cho cài đặt assembly vào GAC.

Trong chốc lát bạn sẽ làm quen với công cụ giúp tự động hóa các bước vừa kể trên. Trong tạm thời, bạn nên tìm hiểu chính xác làm thế nào COM client tương tác với các kiểu dữ liệu .NET sử dụng CCW.

## 7.7.1 Vai trò của CCW (COM Callable Wrapper)

Khi COM Client truy cập một .NET type, CLR thường dùng một proxy được mang tên là CCW (COM Callable Wrapper) để thương lượng việc chuyển đổi từ COM qua .NET. Xem hình 7-30.

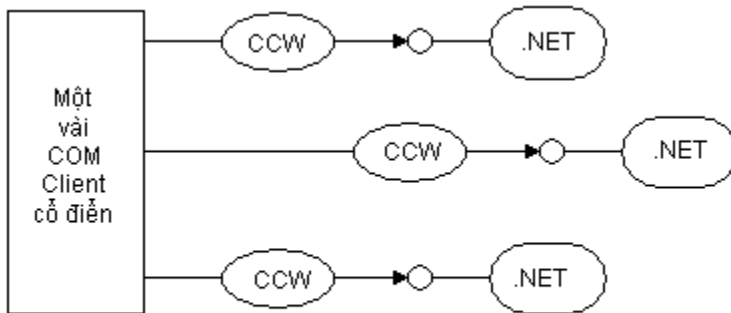
CCW được xem như là một thực thể quản lý cái đếm qui chiếu (reference counter). Điều này là hợp lý, vì rằng COM client giả định là CCW là một kiểu dữ liệu COM thật sự và do đó sẽ tuân thủ các qui tắc của **AddRef()** và **Release()**. Khi COM client phát ra release cuối cùng, CCW sẽ release qui chiếu của nó về kiểu dữ liệu .NET thật sự, và vào thời điểm này sẵn sàng cho phép GC (garbage collector) thu hồi ký ức.

CCW thiết lập tự động một số COM interface làm cho có cảm tưởng proxy tượng trưng cho một coclass. Ngoài tập hợp những custom interface được định nghĩa bởi kiểu dữ liệu .NET (bao gồm một thực thể mang tên class interface bạn sẽ làm quen trong chốc lát) CCW còn cung cấp hỗ trợ đối với các cách hành xử COM được mô tả trên bảng 7-9.

**Bảng 7-19: CCW hỗ trợ vô số giao diện cốt lõi COM**

Các giao diện thi công bởi CCW	Mô tả
<b>IConnectionPointContainer</b> <b>IConnectionPoint</b>	Nếu .NET type chịu hỗ trợ bất cứ tình huống nào, các giao diện này tượng trưng cho các COM connection point.
<b>IEnumVariant</b>	Nếu .NET chịu hỗ trợ giao diện <b>IEnumerable</b> , thì giao diện này xuất hiện đối với COM client như là một COM enumerator chuẩn.
<b>ISupportErrorInfo</b>	Các giao diện này cho phép các coclass chuyển đi các đối

<b>IErrorInfo</b>	tượng COM error.
<b>ITypeInfo</b> <b>IProvideClassInfo</b>	Các giao diện này cho phép COM client xem như mình thao tác thông tin kiểu dữ liệu COM của một assembly. Trong thực tế, COM client tương tác với .NET metadata.
<b>IUnknown</b> <b>IDis[atch</b> <b>IDispatchEx</b>	Các giao diện cốt lõi này cung cấp việc hỗ trợ đối với kết nối sớm và kết nối trễ đối với .NET type. IDispatchEx có thể được hỗ trợ bởi CCW nếu .NET type thì công giao diện IExpando.



**Hình 7-30: COM type nói chuyện với .NET type sử dụng một CCW**

## 7.7.2 Tìm hiểu Class Interface

Trên COM cổ điển, cách duy nhất một COM client có thể liên lạc với một đối tượng COM là sử dụng một qui chiếu giao diện (interface reference). Tuy nhiên, một vài ngôn ngữ ăn ý với COM (chẳng hạn Visual Basic 6.0) thường che dấu việc này khỏi lập trình viên làm cho Visual Basic có vẻ dễ lập trình hơn. Thật ra, ở hậu trường Visual Basic kết sinh một [default] interface đối với mỗi coclass trong COM binary. Bất cứ thành viên nào được khai báo là Public sẽ được đưa vào [default] interface. Theo cách này, lập trình viên Visual Basic có cảm tưởng mình làm việc với một qui chiếu đối tượng, nhưng thật ra họ làm việc với một qui chiếu giao diện như sau:

```
` Xin nhớ cho Visual Basic che dấu [default] interface
Dim o As New MyComClass      `Query đối với [default] _MyComClass
o.Hello                      ` Thật sự triệu gọi _MyComClass->Hello().
```

Ngược lại, các kiểu dữ liệu .NET không cần phải hỗ trợ bất cứ giao diện nào. Ta có thể xây dựng một giải pháp tròn vẹn bằng cách chỉ dùng qui chiếu đối tượng mà thôi. Tuy nhiên, vì rằng COM client cổ điển không làm việc được với qui chiếu đối tượng, một trách nhiệm khác của CCW là phải hỗ trợ một class interface để tượng trưng cho mỗi thuộc tính, hàm hành sự, vùng mục tin và tình huống được định nghĩa bởi public sector của kiểu dữ liệu. Như bạn có thể thấy, CCW tiếp cận vấn đề cũng giống như Visual Basic 6.0.

## 7.7.2.1 Định nghĩa một Class Interface

Attribute **ClassInterface** là tùy chọn nhưng là một kiểu dữ liệu rất quan trọng. Theo mặc nhiên, bất cứ hàm hành sự nào được định nghĩa trên một .NET class sẽ được trưng ra cho COM như là một dispinterface thô (nghĩa là một thi công nào đó của IDispatch). Như vậy, tất cả các COM client nào muốn sử dụng các hàm hành sự cấp lớp phải thi hành late binding để thao tác các kiểu dữ liệu .NET của bạn. Để bỏ qua các hành xử mặc nhiên này, bạn dùng kiểu dữ liệu **ClassInterfaceAttribute**, được gán cho bất cứ trị nào của enumeration **ClassInterfaceType**, như theo bảng 7-10 sau đây:

**Bảng 7-10: Các trị của ClassInterfaceType Enumeration**

Trị enum	Mô tả
<b>AutoDispatch</b>	Cho biết là một giao diện dispatch-only sẽ được kết sinh đối với lớp.
<b>AutoDual</b>	Cho biết một giao diện dual sẽ được kết sinh đối với lớp.
<b>None</b>	Cho biết không giao diện nào được kết sinh đối với lớp.

Trong thí dụ kế tiếp, bạn sẽ khai báo **ClassInterfaceType.AutoDual** cho biết loại giao diện lớp. Theo cách này, late binding client (chẳng hạn VBScript) có thể truy cập các hàm hành sự **Add()** và **Subtract()** sử dụng **IDispatch**, trong khi early binding client (chẳng hạn Visual Basic và C++) có thể sử dụng class interface (mang tên **\_CSharpCalc**). Giống như trên Visual Basic 6.0, tên của class interface của bạn bao giờ cũng dựa trên tên kiểu dữ liệu và một dấu '\_' nằm ở đầu.

## 7.7.3 Xây dựng .NET type của bạn

Để minh họa một kiểu dữ liệu COM liên lạc với managed code, giả sử bạn tạo một C# Class Library đơn giản mang tên **DotNetClassLib**, chịu hỗ trợ hai hàm hành sự **Add()** và **Subtract()**. Ngoài ra, cũng giả sử luôn bạn có định nghĩa (và thi công) một giao diện khác mang tên **IAdvancedMath** cho phép nhân và chia. Phần logic khá đơn giản. Tuy nhiên, bạn để ý việc sử dụng attribute **ClassInterface**, như sau:

**Thí dụ 7-11: Một C# Class Library: DotNatClassLib.**

\*\*\*\*\*

```
using System;
using System.Runtime.InteropServices;

namespace DotNetClassLib
{
    public interface IAdvancedMath
    {
        int Multiple(int x, int y);
        int Divide(int x, int y);
    }
    [ClassInterface(ClassInterfaceType.AutoDual)]
    public class CSharpCalc: IAdvancedMath
```

```

{   public CSharpCalc() {}
    public int Add(int x, int y) {return x+y;}
    public int Subtract(int x, int y) {return x-y;}
    int IAdvancedMath.Multiple(int x, int y) {return x*y;}
    int IAdvancedMath.Divide(int x, int y)
    {   if(y == 0)
        // chặn húng như là đối tượng COM error
        throw new DivideByZeroException();
        return x/y;
    }
}
}
}
*****

```

## 7.7.4 Kết sinh Type Library và cho đăng ký .Net Type

Một khi bạn đã biên dịch xong dự án, bạn có thể có hai cách tiếp cận để kết sinh thông tin kiểu dữ liệu (type information) và cho đăng ký assembly vào system registry. Tiếp cận đầu tiên là sử dụng trình tiện ích **regasm.exe** có trong .NET SDK. Chức năng mặc nhiên của công cụ này là việc đăng ký COM cần thiết vào hệ thống và cho phép COM SCM tìm thấy nơi tá túc và nạp assembly theo yêu cầu COM client. Tuy nhiên, nếu bạn khai báo flag /tlb, thì công cụ này cũng kết sinh type library được yêu cầu, như sau:

```
regasm DotNetClassLib.dll /tlb:simpledotnetserver.tlb
```

Một cách tiếp cận thay thế khác là bạn có thể dùng **regasm.exe** để cho đăng ký đúng thông tin vào system registry và kết sinh type info sử dụng một công cụ riêng biệt mang tên **tlbexp.exe**.

```
regasm DotNetClassLib.dll
```

rồi tiếp

```
tlbexp DotNetClassLib.dll /out:simpledotnetserver.tlb
```

Cả hai trường hợp đều đi đến cùng kết quả là .NET assembly của bạn được cấu hình trên system registry và bạn có một COM type library mô tả nội dung của assembly.

## 7.7.5 Quan sát Exported Type Information

Bạn đã kết sinh COM type library tương ứng, bạn có thể sử dụng OLE/COM Object Viewer trên IDE bằng cách nạp tập tin \*.tlb. Từ đó bạn có thể thấy phần định nghĩa IDL đối với CSharpCalc class interface (**\_CSharpCalc**) như sau:



```
[
    odl,
    uuid(AA165958-53F3-3129-83AE-7AE174FE923F),
    hidden, dual, nonextensible, oleautomation,
    custom(0F21F359-AB84-41E8-9A78-36D110E6D2F9,
           DotNetClassLib.CSharpCalc)
]
interface _CSharpCalc: IDispatch
{
    // Các hàm System.Object
    [id(00000000), propget,
     custom(54FC8F55-38DE-4703-9C4E-250351302B1C, 1)]
    HRESULT ToString([out, retval] BSTR* pRetVal);
    [id(0x60020001)]
    HRESULT Equals(
        [in] VARIANT obj,
        [out, retval] VARIANT_BOOL* pRetVal);
    [id(0x60020002)]
    HRESULT GetHashCode([out, retval] long* pRetVal);
    [id(0x60020003)]
    HRESULT GetType([out, retval] _Type** pRetVal);

    // Các hàm của class interface
    [id(0x60020004)]
    HRESULT Add([in] long x, [in] long y,
               [out, retval] long* pRetVal);
    [id(0x60020005)]
    HRESULT Subtract([in] long x, [in] long y,
                    [out, retval] long* pRetVal);
};
```

Như được khai báo bởi attribute **ClassInterfaceAttribute**, [default] được cấu hình như một [dual]. Như bạn có thể thấy, class interface cũng liệt kê rõ ra đối với các thành viên của **System.Object**.

Một điểm đáng lưu ý là class interface *không hỗ trợ* các thành viên của giao diện **IAdvancedMath**. Lý do là vì giao diện này được thi công sử dụng explicit interface implementation. Do đó, type library được kết sinh cũng có chứa một định nghĩa IDL đối với custom interface này như sau:

```
interface IAdvancedMath: IDispatch
{
    [id(0x60020000)]
    HRESULT Multiple([in] long x, [in] long y, [out, retval]
                    long* pRetVal);
    [id(0x60020001)]
    HRESULT Divide([in] long x, [in] long y, [out, retval]
                  long* pRetVal);
};
```

Nếu bạn không sử dụng explicit interface inheritance, bạn cũng sẽ có một định nghĩa đứng một mình đối với `IAdvancedMath`. Tuy nhiên, bạn cũng có thể thấy class interface mặc nhiên sẽ được điền đầy bởi các thành viên `Multiply()` và `Divide()`.

### 7.7.5.1 *\_Object Interface*

IDL được kết sinh còn có chứa một giao diện mang tên **\_Object**. Giao diện này tượng trưng cho một biểu diễn unmanaged của **System.Object**. Như vậy, các kiểu dữ liệu COM nào tiêu thụ những .NET type có thể sử dụng những thành viên cốt lõi của lớp cơ bản này.

Định nghĩa IDL coclass tự động thêm hỗ trợ đối với interface type này:

```
coclass CSharpCalc
{
    [default] interface _CSharpCalc;
    interface _Object;
    interface IAdvancedMath;
};
```

Trong chốc lát bạn sẽ thấy giao diện `_Object` vào cuộc.

### 7.7.5.2 *Câu lệnh Library được kết sinh*

Điểm cuối cùng đáng quan tâm đối với thông tin kiểu dữ liệu được kết sinh là cấu hình của câu lệnh library. Trên COM cổ điển, câu lệnh library được dùng tượng trưng cho mỗi IDL type cần được đưa vào tập tin \*.tlb. Tập tin này chẳng qua cũng chỉ là một binary tương đương của IDL và là chìa khoá cho sự độc lập ngôn ngữ của COM (và giữ vai trò chính trong việc marshalling các kiểu dữ liệu xuyên ranh giới). Các qui tắc dùng bởi **tlbexp.exe** rất đơn giản. Các namespace .NET sẽ được lấp đầy dựa trên câu lệnh library COM.

Bạn đã thấy những định nghĩa của các kiểu dữ liệu .NET (class interface, `IAdvancedMath`, coclass, v.v.). Tuy nhiên, điểm đáng ghi nhớ là ngoài việc nhận khẩu thông tin kiểu dữ liệu OLE chuẩn, câu lệnh thư viện của bạn còn nhập khẩu thông tin kiểu dữ liệu mô tả `mscorlib.dll` (phần .NET base class library assembly cốt lõi) và runtime execution engine (`mscorlib.dll`) hoặc OLE Automation:

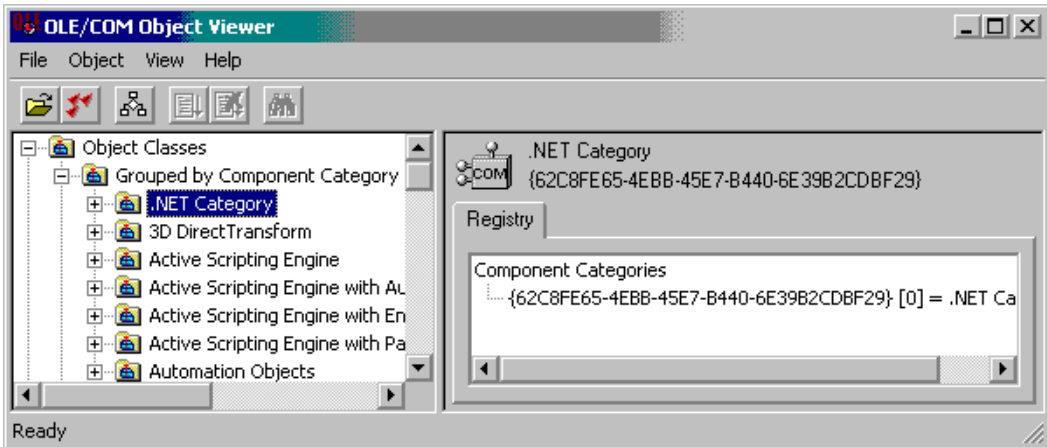
```
[
    uuid(C2AAF144-D3DF-3D8E-B46E-D006869D3BBC), version(1.0),
    custom(90883F05-3D28-11D2-8F17-00A0C9A6186D, DotNetClassLib,
    Version=1.0.1486.4394, Culture=neutral, PublicKeyToken=null)
]
```

```

library DotNetClassLib
{
    // TLib:
    // TLib: Common Language Runtime Library:
        {BED7F4E7-1A96-11D2-8F08-00A0C9A6186D}
    importlib("mscorlib.tlb");
    // TLib: OLE Automation: {00020430-0000-0000-C000-000000000046}
    importlib("stdole2.tlb");
}

```

## 7.7.6 Nhìn xem kiểu dữ liệu sử dụng OLE/COM Object Viewer



Hình 7-31: Tất cả các assembly được đăng ký đều thuộc thành phần .NET Category

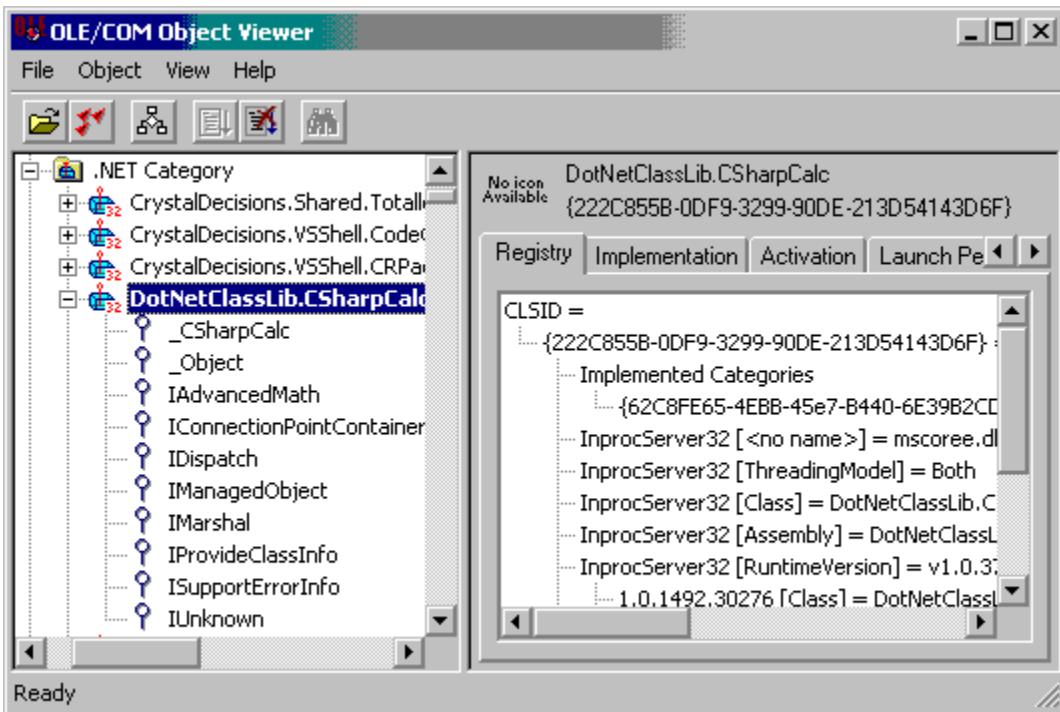
COM hỗ trợ khái niệm về các loại cấu kiện (component categorie). Đây là một GUID (được gọi là CATID) nhận diện một tập hợp các coclass có liên hệ với nhau. OLE/COM Object Viewer cho phép bạn rà qua các CATID đã được đăng ký trên máy tính của bạn từ một GUI Tree View control (hình 7-31)

Nếu bạn tìm ProgID đối với assembly DotNetServerLib, bạn để ý là khi bạn muốn bung node này, bạn sẽ nhận một thông điệp sai lầm. Bạn nhớ lại là khi một COM client muốn sử dụng một assembly .NET, assembly phải nằm cùng thư mục với ứng dụng khởi động (hoặc được cài đặt trong GAC).

Để điều chỉnh vấn đề bạn cho chép assembly simpleDotNetServer.tlb vào thư mục cùng với OLE/COM Object Viewer, là thư mục dự án DotNetClassLib (hoặc cài đặt assembly vào GAC). Một khi làm xong việc này, bạn có khả năng xem xét các giao diện mà proxy hỗ trợ (hình 7-32)

## 7.7.7 Quan sát các mục vào Registry

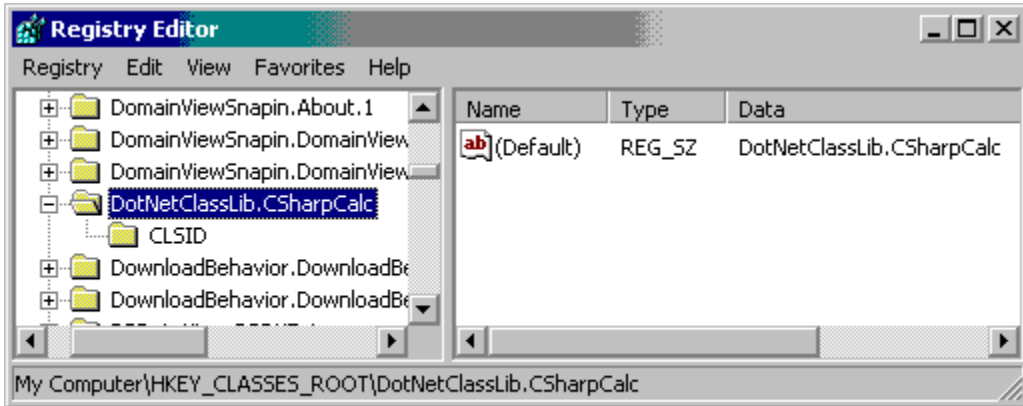
Chi tiết cuối cùng phải xem xét trước khi sử dụng assembly từ COM client là các mục vào trên Registry được đưa vào bởi trình tiện ích **regasm.exe**. Trước tiên, bạn sẽ nhận mã nhận diện ProgID cho mỗi coclass được định nghĩa trong assembly (hình 7-33). Từ ProgID, bạn có thể rảo qua mục kế tiếp mà bạn quan tâm, CLSID (mã nhận diện lớp). Xem hình 7-34.



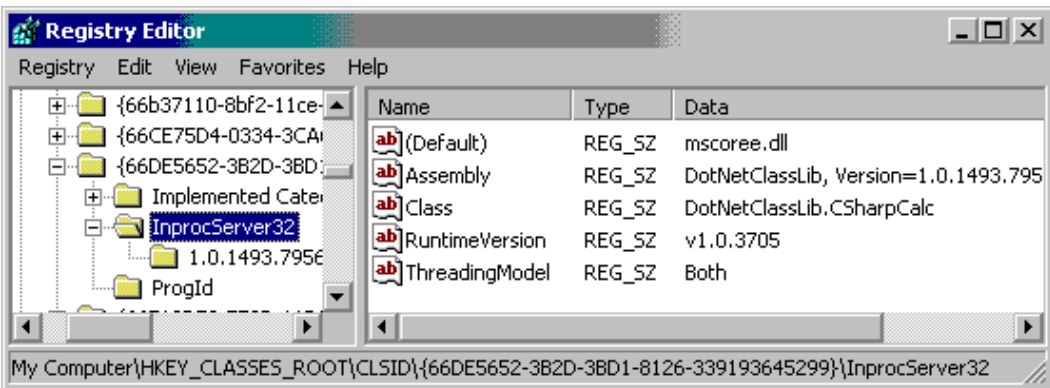
**Hình 7-32: Giao diện được thi công bởi CCW**

Khi một COM client đưa ra một yêu cầu khởi động đối với SCM, thì SCM sẽ đáp ứng bằng cách tham khảo HKCR\CLSID để tìm ra nơi tá túc của server được đăng ký. Thư mục con quan trọng nhất là InprocServer32, lo trừu lồi tìm về binary mà SCM sẽ nạp cho COM client. Tuy nhiên, bạn sẽ không tìm thấy một danh sách đối với DotNetClassLib.dll (vì nó không phải là một COM server). Thay vào đó, bạn sẽ tìm thấy lồi tìm về CLR execution engine (hình 7-35).

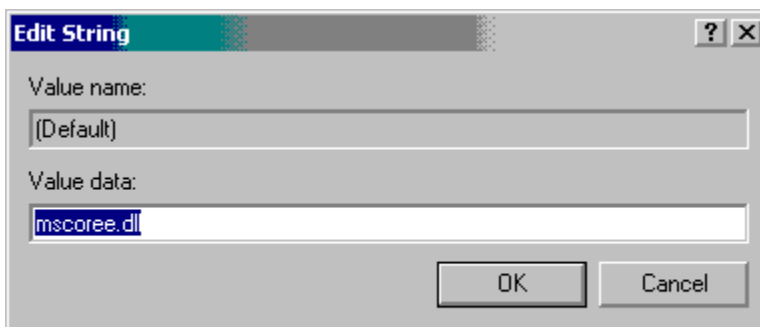
Ngoài ra, cũng ở thư mục con InprocServer32 có một mục vào mới mang tên Assembly, cho biết tên chính danh (fully quantified name) của assembly. Xem hình 7-36.



Hình 7-33: Mã nhận diện ProgID được đăng ký



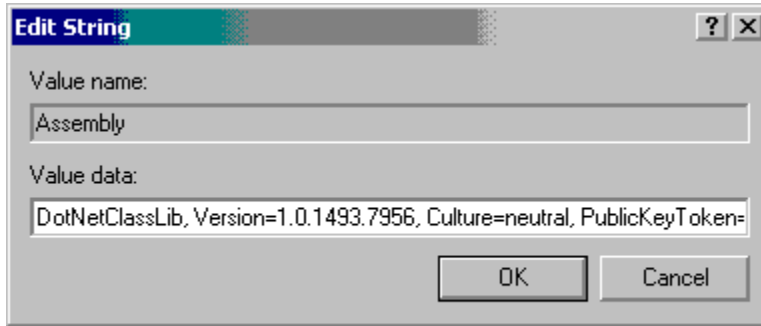
Hình 7-34: CLSID được đăng ký



Hình 7-35: InprocServer32 chỉ về .NET execution engine

được đăng ký dưới HKCR\TypeLib. Kết quả cuối cùng là COM SCM có thể nạp và thao tác lên nội dung của assembly .NET. Sau hậu trường, CLR execution engine nạp assembly, xây dựng CCW và điều khiển trò chơi.

Lẽ dĩ nhiên, có những mục vào khác được đưa vào bởi **regasm.exe**. Thí dụ, biết rằng các kiểu dữ liệu .NET nào đó bao giờ cũng đi qua các giao diện được dẫn xuất từ IDispatch, mỗi giao diện sẽ được cấu hình sử dụng `oleaut32.dll`. Type information



Hình 7-36:: Trị của assembly entry

## 7.7.8 Xây dựng một Visual Basic 6.0 Test Client

Tới đây bạn đã có một assembly .NET được cấu hình tương tác với một COM runtime, bạn có thể tạo một vài COM client, chẳng hạn một dự án Visual Basic 6.0 Standard EXE đơn giản và cho qui chiếu về type library mới được kết sinh. Trước khi bạn thêm vào bất cứ đoạn mã nào, bạn cho cất trữ toàn bộ dự án lên đĩa tại một nơi nào đó dễ nhận ra. Sau đó, bạn làm một bản sao assembly C# rồi đưa nó vào cùng thư mục với COM client (hoặc cho cài đặt lên GAC).

Để cho đơn giản phần GUI chỉ gồm một nút <Use .NET Object> dùng thao tác đối tượng .NET. Tuy nhiên, bạn còn nhớ khi ta tạo C# calculator, bạn đã cấu hình hàm hành sự **Divide()** của giao diện **IAdvancedMath** là gọi đi một biệt lệ **DivideByZero Exception**. Do đó, bạn có thể chặn hứng biệt lệ .NET này như là một đối tượng COM error. Sau đây là đoạn mã Visual Basic 6.0, thí dụ 7-12. Bạn để ý là ta dùng một vài hàm hành sự được kế thừa được định nghĩa bởi giao diện `_Object`:

### Thí dụ 7-12: Visual Basic 6.0 COM client

\*\*\*\*\*

```
Private Sub btnDoEverything_Click()
    On Error GoTo OOPS:

    // Tạo .NET type và thêm vài con số
    Dim o As New CSharpCalc
    MsgBox o.Add(30, 30), , "Adding"

    // triệu gọi vài thành viên của _Object
    MsgBox o.ToString, , "To String"
    MsgBox o.GetHashCode, , "Hash code"

    Dim t As Object
    Set t = o.GetType()
    MsgBox t, , "Type"
```

```
// Đi lấy một giao diện mới (và kích hoạt biệt lệ)
Dim i As IAdvancedMath
Set i = o
MsgBox i.Multiple(4, 22), , "Multiply"
MsgBox i.Divide(20, 2), , "Divide"
MsgBox i.Divide(20, 0) ' Throw error.

OOPS:
MsgBox Err.Description, , "Error!" // In ra biệt lệ
End Sub
*****
```

Bạn để ý Visual Basic 6.0 không cho phép bạn truy cập vài giao diện `_Type` do `_Object.GetType()` trả về, vì nó được đánh dấu như là [hidden] interface. Cách tốt nhất đối với bạn là đưa nó vào một biến đối tượng chung chung (như vậy không lý thú chi mấy).

## 7.7.9 Các vấn đề do .NET-to-COM đặt ra

Bao giờ bạn cũng nên nhớ cho CTS (Common Type System) định nghĩa một số cấu trúc không thể đơn giản được tương ứng đối với COM classic. Thí dụ, C# class type có thể hỗ trợ bất cứ hàm constructor nào, cũng như các tác tử và hàm hành sự nạp chồng và có thể dẫn xuất với nhau sử dụng kế thừa lớp. Theo COM cổ điển thì các khái niệm cơ bản này không hiểu được. Do đó, ở hậu trường, bạn có thể hình dung là **tlbexp.exe** phải làm gì đó để xây dựng COM type library.

Muốn hiểu được vấn đề bạn cần tạo một test case. Giả sử một dự án C# code library chứa một lớp cơ bản và một lớp được dẫn xuất (thí dụ 7-13). Lớp cơ bản, **BaseClass**, định nghĩa một vài vùng mục tin, một hàm constructor và một hàm virtual đơn côi, còn lớp được dẫn xuất, **DerivedClass**, sẽ phủ quyết (override) thành viên virtual và khai báo một hàm hành sự nạp chồng, như sau:

### Thí dụ 7-13: Một đoạn mã library C#

```
*****
[ClassInterface(ClassInterfaceType.AutoDual)]
public class BaseClass
{
    // dữ liệu trạng thái
    private int memberVar;
    public string fieldOne;
    // hàm constructor
    public BaseClass() {}
    public BaseClass(int m, string f) {memberVar = m; fieldOne = f;}
    // hàm hành sự virtual
    public virtual void VirMethod()
    { Console.WriteLine("Base VirMethod impl"); }
}
```

```
[ClassInterface(ClassInterfaceType.AutoDual)]
public class DerivedClass: BaseClass
{
    // State data
    public float fieldTwo;
    // hàm constructor
    DerivedClass(int m, string f):base(m, f) {}
    // hàm hành sự bị phủ quyết
    public override void VirMethod()
    {
        Console.WriteLine("Derived VirMethod impl");
        base.VirMethod();
    }
    // Các thành viên nạp chồng
    public void SomeMethod() {}
    public void SomeMethod(int x) {}
    public void SomeMethod(int x, object o) {}
    public void SomeMethod(int x, float f) {}
}
*****
```

Rõ ràng là bạn không bận tâm các hàm hành sự này làm gì. Bạn chỉ quan tâm làm thế nào trình tiện ích **tlbexp.exe** sẽ ánh xạ những kiểu dữ liệu .NET này qua các hàm bấm sinh COM.

### 7.7.9.1 Quan sát thông tin kiểu dữ liệu của lớp *BaseClass*

Trước tiên, thử xem xét định nghĩa coclass đối với managed BaseClass type (nạp tập tin \*.tlb vào OLE/COM Object Viewer để kiểm tra) cho thấy như sau:

```
coclass BaseClass {
    interface IManagedObject;
    [default] interface _BaseClass;
    interface _Object;
};
```

Chỉ có chi bàn cãi ở đây vì bạn đã hiểu khái niệm về một class interface và vai trò của định nghĩa giao diện **\_Object**. Phần quan trọng nằm ở ngay phần định nghĩa class interface, như sau:

```
interface _BaseClass: IDispatch
{
    // Các hàm hành sự _Object ở đây ...

    [id(0x60020004)] HRESULT VirMethod();
    [id(0x60020005), propget,
        HRESULT fieldOne([out, retval] BSTR *pRetVal);

    [id(0x60020005), propput,
        HRESULT fieldOne([in] BSTR pRetVal);
};
```



Bạn để ý là vùng mục tin public được biểu diễn như là thuộc tính COM. Điều này cũng khá hợp lý vì COM client không bao giờ có một qui chiếu cấp đối tượng và bị bắt buộc làm việc với một kiểu dữ liệu trên cấp giao diện với giao diện (interface-by-interface level). Bây giờ ta qua coi tiếp các lớp được dẫn xuất.

### 7.7.9.2 *Quan sát thông tin kiểu dữ liệu của lớp DerivedClass*

Vì COM cổ điển không hỗ trợ kế thừa giữa các kiểu dữ liệu nên rõ ràng **tlbexp.exe** không thể mô hình hóa mối liên hệ is-a giữa kiểu dữ liệu cơ bản và dẫn xuất. Tuy nhiên, bạn nhận được điều tốt lành là thi công giao diện như sau:

```
coclass DerivedClass {
    interface IManagedObject;
    [default] interface _DerivedClass;
    interface _BaseClass;
    interface _Object;
};
```

Bạn để ý kiểu dữ liệu dẫn xuất sẽ thi công class interface của cha-mẹ nó. Theo cách này, kiểu dữ liệu dẫn xuất có thể mang chức năng tương đương với kiểu dữ liệu lớp cơ bản. Class interface của DerivedClass cũng đáng quan tâm. Bạn nhớ lại là thi công managed của kiểu dữ liệu này chịu hỗ trợ một thành viên nạp chồng đơn độc. Vì rằng COM không hỗ trợ cấu trúc cú pháp này, trình tiện ích **tlbexp.exe** đưa ra giải pháp sau đây:

```
interface _DerivedClass: IDispatch
{ // Các hàm hành sự _Object ở đây ...

    // Các hàm hành sự 'kế thừa' của kiểu dữ liệu base
    [id(0x60020004)] HRESULT VirMethod();
    [id(0x60020005), propget,
        HRESULT fieldOne([out, retval] BSTR *pRetVal);
    [id(0x60020005), propput,
        HRESULT fieldOne([in] BSTR pRetVal);
    // Các hàm hành sự nạp chồng
    [id(0x60020007)] HRESULT SomeMethod();
    [id(0x60020008)] HRESULT SomeMethod_2([in] long x);
    [id(0x60020009)] HRESULT SomeMethod_3([in] long x,
        [in] VARIANT o);
    [id(0x6002000a)] HRESULT SomeMethod_4([in] long x,
        [in] single f);
    // Vùng mục tin
    [id(0x6002000b), propget,
        HRESULT fieldTwo([out, retval] single *pRetVal);
    [id(0x6002000b), propput,
        HRESULT fieldTwo([in] single pRetVal);
};
```

Ở đây, bạn thấy cách đánh số đuôi dùng cho các hàm hành sự nạp chồng. Các cách ánh xạ khác mà bạn sẽ gặp phải bao gồm namespace nằm lồng, các lớp cơ bản trừu tượng, trị kiểu dữ liệu (enum và struct), và v.v.. (bạn có thể cho chuyển đổi assembly **CarLibrary.dll** mà bạn đã xây dựng trước đây thông qua trình tiện ích **tlbexp.exe** để tăng cường sự hiểu biết của mình).

## 7.8 Điều khiển phần Generated IDL (hoặc ảnh hưởng lên TlbExp.exe)

Như bạn có thể thấy, khi bạn sử dụng trình tiện ích **tlbimp.exe** để tạo một proxy, metadata được kết sinh sẽ tự động được thêm nhiều attribute khác nhau. Khi bạn xây dựng những kiểu dữ liệu .NET mà bạn mong đợi sẽ được sử dụng bởi COM client cổ điển, bạn có thể dùng trực tiếp các attribute này (chẳng hạn **ClassInterfaceAttribute** type) trong managed code. Diễn hình, điều này chỉ có thể làm để phủ quyết ánh xạ mặc nhiên mà trình tiện ích đã tạo ra.

Để minh họa tiến trình điều khiển đối với COM Type Information được kết sinh, sau đây là một namespace mới (mang tên **AttribDotNetObjects**), thí dụ 7-14, định nghĩa một giao diện duy nhất (**IBasicMath**) cũng như một class type duy nhất (**Calc**). Bạn để ý là ta sẽ dùng những attribute khác nhau để điều khiển GUID của kiểu dữ liệu được kết sinh cũng như định nghĩa nằm sau của giao diện **IBasicMath** và hàm hành sự **Add()**. Ngoài ra, bạn cũng để ý cho là ở đây lớp **Calc** định nghĩa hai hàm static mang thêm những attribute đặc thù.

### Thí dụ 7-14: Một namespace mới với các attribute

```
*****
using System;
using System.Runtime.InteropServices;
using System.Windows.Forms;

// Giao diện .NET này mang một số attribute khác nhau mà
// trình tiện ích tlbimp.exe sẽ dùng đến.
namespace AttribDotNetObjects
{
    [GuidAttribute("47430E06-718D-42C6-9E45-78A99673C43C"),
     InterfaceTypeAttribute(ComInterfaceType.InterfaceIsDual)]
    public interface IBasicMath
    {
        [DispId(777)] int Add(int x, int y);
    }
    [GuidAttribute("C08F4261-C0C0-46AC-87F3-EDE306984ACC")]
    public class DoNetCalc: IBasicMath
    {
        public void DotNetCalc(){}
    }
}
```

```

public int Add(int x, int y) {return x+y;}

[ComRegisterFunctionAttribute]
public static void AddExtraRegLogic(string regLoc)
{
    MessageBox.Show("Inside AddExtraRegLogic f(x)",
                    ".NET assembly says:");
}

[ComUnregisterFunctionAttribute]
public static void RemoveExtraRegLogic(string regLoc)
{
    MessageBox.Show("Inside RemoveExtraRegLogic f(x)",
                    ".NET assembly says:");
}
}
}
*****

```

## 7.8.1 Quan sát thông tin kiểu dữ liệu COM được kết sinh (IDL)

Nếu bạn cho chạy assembly sử dụng trình tiện ích **tlbexp.exe**, sau đó bạn sử dụng OLE/COM Object Viewer, bạn sẽ thấy các mã nhận diện IID và CLSID mang cùng trị như được liệt kê dưới đây, và ngoài ra giao diện IBasicMath cũng được cấu hình như là [dual]. Sau đây là phần IDL đối với giao diện IBasicMath.

```

// Trong assembly chúng tôi viết các attribute sau đây:
// [GuidAttribute("47430E06-718D-42C6-9E45-78A99673C43C"),
// InterfaceTypeAttribute(ComInterfaceType.InterfaceIsDual)]

// IDL được kết sinh
[ odl,
 uuid(47430E06-718D-42C6-9E45-78A99673C43C),
 version(1.0), dual, oleautomation,
 custom(0F21F359-AB84-41E8-9A78-36D110E6D2F9,
  AttribDotNetObjects.IBasicMath)
]
interface IBasicMath: IDispatch
{
    [id(0x00000309)] // chúng tôi viết [DispId(777)]
    HRESULT Add([in] long x, [in] long y,
        [out, retval] long* pRetVal);
};

```

Nếu bạn trở lại cách biểu diễn .NET của giao diện IBasicMath và nhật tu lô gic của hàm constructor InterfaceTypeAttribute() như sau:

```
// Làm cho nó thành một custom interface (chứ không phải dual)
[GuidAttribute("47430E06-718D-42C6-9E45-78A99673C43C"),
InterfaceTypeAttribute(ComInterfaceType.InterfaceIsDual)]
public interface IBasicMath
{
    int Add(int x, int y);
}
```

Bạn sẽ thấy định nghĩa IDL sẽ như sau một khi trở về từ trình tiện ích tlbexp.exe:

```
// IDL được kết sinh, không còn là [dual]
[ odl,
  uuid(47430E06-718D-42C6-9E45-78A99673C43C) ,
  oleautomation,
  custom(0F21F359-AB84-41E8-9A78-36D110E6D2F9,
    AttribDotNetObjects.IBasicMath)
]
interface IBasicMath: IUnknown
{
    // không còn DispId
    HRESULT _stdcall Add([in] long x, [in] long y,
        [out, retval] long* pRetVal);
};
```

## 7.8.2 Tương tác với Assembly Registration

Tiếp theo, bạn cần xem xét việc sử dụng các kiểu dữ liệu **ComRegisterFunction Attribute** và **ComUnregisterFunctionAttribute**. Như bạn đã biết, COM server cho xuất khẩu hai hàm hành sự (**DllRegisterServer** và **DllUnregisterServer**) mà các trình tiện ích khác nhau sẽ triệu gọi để chèn (hoặc gỡ bỏ) thông tin đăng ký COM được yêu cầu. .NET binary không xuất khẩu những hàm như thế. Tuy nhiên, bằng cách khai báo những hàm hành sự static với những attribute kể trên, bạn có thể mô phỏng cách hành xử tương tự. Để minh họa, nếu bây giờ bạn cho đăng ký .NET assembly sử dụng regasm.exe:

```
regasm AttribDotNetObjects.dll
```



Hình 7-37: Tương tác với COM Registration

Bạn sẽ thấy một message box, như theo hình 7-37:

Cuối cùng một vài điểm cần lưu ý đối với hai attribute đăng ký này. Thứ nhất, tên của hàm hành sự không tạo khác biệt gì mấy. Tuy nhiên, tên phải là một đối mục kiểu chuỗi duy nhất cho biết nơi tá túc hiện hành của registry được nhật tu. Ngoài ra, nếu bạn cấu hình hóa một hàm hành sự static mang **ComRegisterFunction**

**Attribute**, bạn cũng phải cấu hình hóa một hàm hành sự mang attribute **ComRegisterFunctionAttribute**. Theo cách này, bạn có thể mô phỏng một COM server tự đăng ký lấy.

## 7.9 Tương tác với COM+ Services

Mục chót của chương này là khảo sát làm thế nào các base class library cho phép xây dựng những kiểu dữ liệu .NET có thể được cấu hình để lợi dụng những lợi thế của lớp COM+ runtime. Trước khi minh họa tiến trình tổng quát, chúng tôi bắt đầu cái nhìn tổng quan cấp cao liên quan đến vai trò của COM+.

Có thể bạn đã biết qua một phần mềm mang tên MTS (Microsoft Transaction Server). MTS là một ứng dụng server cho bạn khả năng cho lưu trữ (host) các COM DLL cổ điển phù hợp với cấp xí nghiệp trong một môi trường “đa-nguyên” n-tier. Thí dụ, giả sử bạn đã tạo ra một COM binary cổ điển lo kết nối với dữ liệu nguồn (có thể là sử dụng ADO) để nhập tu một số bảng dữ liệu có liên hệ với nhau. Một khi COM server này được cài đặt dưới MTS, nó sẽ kế thừa một số đặc tính cốt lõi, chẳng hạn khai báo giao dịch (declarative transaction), hiện dịch ngay liền (just-in-time, JIT, activation) và ASAP deactivation cũng như mô hình an ninh dựa trên vai trò. Kết quả cuối cùng là bạn có thể cấu hình hóa làm thế nào kiểu dữ liệu chịu chơi MTS phải hành xử theo thể thức khai báo (declarative manner) thay vì theo lệnh lô gic được viết ra (logic hard-coded).

Mỗi COM class chịu chơi MTS sẽ có một đối tượng context được gắn liền dùng trữ một số đặc tính cho biết đối tượng MTS sẽ được dùng thế nào. Thí dụ, context có thể chứa thông tin liên quan đến ủy nhiệm an ninh (security credential) của phía triệu gọi, đến kết xuất giao dịch (transaction outcome) của đối tượng này (nghĩa là happy bit), và đến việc liệu xem đối tượng có sẵn sàng cho thu hồi lại khỏi ký ức (nghĩa là done bit) hay không.

Phần lớn, các MTS COM type được tạo bởi những thực thể stateless (không trạng thái). Điều này đơn giản ngụ ý là đối tượng có thể được tạo ra và hủy bỏ bởi MTS runtime (để thu hồi lại nguồn lực hệ thống) mà không ảnh hưởng gì đến connected base client (nghĩa là thực thể triệu gọi lớp MTS runtime). Như vậy, các kiểu dữ liệu MTS thuộc loại GUI-less và giữ vai trò các đối tượng business cổ điển lo thi hành một đơn vị công tác cho base client rồi lặng lẽ biến đi. Nếu base client triệu gọi đối tượng nó nghĩ rằng nó có qui chiếu về đối tượng MTS này và MTS runtime chỉ đơn giản tạo ra một bản sao mới.

Mặc dù MTS cho phép xây dựng những hệ thống phân tán khá dĩ tăng qui mô rất cao và rất khả tín, như nó cũng có bề mặt khó ưa. Đặc biệt COM và MTS runtime không “hoà hợp” chặt chẽ chi cho lắm. Thí dụ, mỗi kiến trúc viết ra những phân duy nhất của

registry, có thể ngăn không cho COM DLL hoạt động như là một in-proc server. Ngoài ra, cơ chế tạo dựng đối tượng được dùng bởi COM không cùng mô hình mà MTS sử dụng đến. Khi các đối tượng được cài đặt chạy với MTS, các đối tượng này phải tạo ra những kiểu dữ liệu COM mang tính MTS-hosted sử dụng đến một hàm hành sự đặc biệt được hỗ trợ bởi context được gắn liền.

COM+ trong nhiều khía cạnh là một phiên bản MTS được dọn dẹp sạch sẽ thích ứng hơn. Với COM+, COM cổ điển và MTS cổ điển được thống nhất thành một hệ thống duy nhất lo việc đăng ký cũng như tạo dựng các đối tượng một cách nhất quán hơn. Các ứng dụng COM+ vẫn tiếp tục kế thừa những đặc tính cốt lõi của MTS (declarative transaction, role-based security, v.v..) cũng như vài đặc tính bổ sung. Sau đây là một vài hành xử đặc thù COM+:

- **Chịu hỗ trợ object pooling<sup>24</sup>.** Lớp COM+ runtime có thể duy trì một collection các coclass hiện dịch mà ta có thể trao ngay cho base client. Như vậy thời gian base client chờ nhận qui chiếu giao diện sẽ giảm đi nhiều từ kiểu dữ liệu COM+. Tuy nhiên, phía máy tính COM+ server phải tốn thêm ký ức.
- **Một mô hình tình huống mới mang tên LCE (Loosely Coupled Events).** Mô hình LCE (tình huống được gắn kết lỏng lẻo) cho phép COM+ client và COM+ type liên lạc theo cách thức tách rời (disconnected). Đây có nghĩa là một COM+ class nào đó có thể phát ra một tình huống không cần biết trước kẻ nào sẽ tiếp nhận tình huống này. Ngoài ra, một COM+ client có thể tiếp nhận một tình huống mà không cần biết đến ai là người phát đi.
- **Hỗ trợ đối với object construction string.** Vì rằng COM client cổ điển không cho phép client kích hoạt lô gic hàm constructor, COM+ đưa vào một giao diện chuẩn (IObjectConstruct) cho phép coclass có khả năng chuyển đi bất cứ thông số khởi động nào dưới dạng một BSTR (sẽ được phân tích ngữ nghĩa trong lòng bởi kiểu dữ liệu).
- **Khả năng điều khiển hành xử nối đuôi của một COM+ type theo cách khai báo.** MSMQ (Microsoft Message Queue) là một dịch vụ thông điệp cấp xí nghiệp khá rối rắm. COM+ đưa vào QC (Queued Component) cho phép cất giấu những điểm rối rắm khỏi lập trình viên.

Như bạn có thể thấy, các dịch vụ mà COM+ cung cấp có thể đơn giản hóa rất lớn việc triển khai các ứng dụng phân tán (distributed application). Vấn đề duy nhất là các đặc tính này ban đầu chủ yếu dành cho các đối tượng COM cổ điển. Muốn cho phép .NET thụ hưởng những lợi điểm kể trên trong triển khai, các base class library cung cấp nhiều tương đương .NET được định nghĩa trong namespace **System.EnterpriseServices**.

---

<sup>24</sup> Object pooling là nơi tập kết các đối tượng

## 7.9.1 Tìm hiểu về namespace **System.EnterpriseServices**

Muốn xây dựng những kiểu dữ liệu managed có thể được cấu hình hóa hoạt động dưới COM+ runtime, bạn cần trang bị cho các thực thể .NET vô số attribute được định nghĩa trong namespace **System.EnterpriseServices**. Nếu bạn đã kinh qua MTS và COM+ bạn sẽ thấy quen quen những mục được đề cập đến ở bảng 7-11:

**Bảng 7-11: Các kiểu dữ liệu cốt lõi của namespace *System.EnterpriseServices***

Các kiểu dữ liệu	Mô tả
<b>ApplicationActivationAttribute</b>	Cho phép bạn khai báo liệu xem các component được trữ trong assembly sẽ chạy trên process của phía tạo ra (library app) hoặc trên một system process (server app).
<b>ApplicationIDAttribute</b>	Khai báo mã nhận diện ứng dụng ID của assembly (như là GUID)
<b>ApplicationQueuingAttribute</b> <b>InterfaceQueuingAttribute</b>	Được dùng hỗ trợ khả năng QC (Queued Component)
<b>AutoCompleteAttribute</b>	Cho đánh dấu hàm hành sự mang attribute <b>AutoComplete</b> . Nếu hàm kết thúc một cách đẹp đẽ, hàm <b>SetComplete()</b> sẽ tự động được triệu gọi. Nếu một biệt lệ được tung ra khi đang thi hành hàm, thì <b>SetAbort()</b> sẽ tự động được triệu gọi.
<b>ComponentAccessControlAttribute</b>	Cho phép có khả năng kiểm tra an toàn đối với những triệu gọi một component nào đó.
<b>ConstructionEnabledAttribute</b>	Cho phép có khả năng hỗ trợ việc xây dựng đối tượng COM+
<b>ContextUtil</b>	Là hàm hành sự được ưa thích để lấy thông tin liên quan đến phạm trù đối tượng COM+1.0. Kiểu dữ liệu này định nghĩa một số thành viên static cho phép bạn có những thông tin phạm trù tập trung vào COM+
<b>DescriptionAttribute</b>	Cho đặt đề mô tả này lên một assembly (app), component, method hoặc interface.
<b>EventClassAttribute</b> <b>EventTrackingEnabledAttribute</b>	Được dùng tương tác với mô hình tình huống COM+ LCE.
<b>JustInTimeActivationAttribute</b>	Cho on/off JIT activation.
<b>SecurityCallContext</b> <b>SecurityCallers</b> <b>SecurityIdentity</b> <b>SecurityRoleAttribute</b>	Được dùng cho phép các kiểu dữ liệu .NET tương tác với mô hình role-based security được dùng bởi MTS/COM+.
<b>SharedPropertyGroupManager</b> <b>SharedPropertyGroup</b> <b>SharedProperty</b>	Cho phép truy cập vào MTS/COM+ shared property manager (SPM).

<b>TransactionAttribute</b>	Cho khai báo kiểu giao dịch có sẵn đối với đối tượng này. Các trị có thể có lấy từ enumeration TransactionOption.
-----------------------------	---

## 7.10 Xây dựng các kiểu dữ liệu chịu chơi COM+

Muốn tạo một .NET assembly có thể được cho lưu trữ bởi COM+ runtime, bạn cần theo kiểu tiếp cận cookbook để xây dựng những kiểu dữ liệu được trưng ra. Để bắt đầu, mỗi .NET class type sẽ được dẫn xuất từ **System.ServicedComponent**. Lớp cơ bản này cung cấp phần thi công mặc nhiên của MTS interface cổ điển, **IObjectControl** (**Activate()**, **Deactivate()** và **CanBePooled()**). Nếu bạn muốn phủ quyết những thi công mặc nhiên này, bạn hoàn toàn tự do làm điều này.

Một khi bạn thêm bất cứ attribute chuyên về COM+ bổ sung nào vào kiểu dữ liệu .NET, bạn cần cho biên dịch lại assembly. Tuy nhiên, để đặt assembly này dưới quyền kiểm soát của COM Runtime, bạn cần sử dụng đến một trình tiện ích mới mang tên **regsvcs.exe**. Công cụ này chịu trách nhiệm đối với một số bước ngoài việc cài đặt kiểu dữ liệu của bạn vào COM+ catalog.

Cuối cùng và có thể là quan trọng nhất, bạn phải cài đặt assembly của bạn vào GAC (xem lại chương 3, “Tìm hiểu về Assembly và cơ chế Version”). Lý do rất đơn giản. Vì rằng trình tiện ích **dllhost.exe** (COM+ surrogate) cần biết nơi tá túc của assembly của bạn để cho nó lưu trữ vào một hoạt động nào đó, nó cần có khả năng biết nơi cư trú của binary. Trong trường hợp này việc cho cài đặt lên GAC là điều hợp lý nhất.

Một khi bạn đã thực hiện những bước vừa kể trên, bạn có thể xây dựng bất cứ số lượng base client nào. Bây giờ ta thử xem một thí dụ.

### 7.10.1 Xây dựng một COM+ Aware C# Type

Để minh họa làm thế nào xây dựng một .NET type có thể đem dùng trên COM+ Runtime, bạn thử xây dựng một managed code library mới mang tên **DotNetCOMPlusServer**. Bạn nên nhớ COM+ Runtime chỉ lưu trữ những kiểu dữ liệu được trữ trong DLL. Bạn cho cấu hình lớp đơn độc (ComPlusType) với những thuộc tính COM+ như sau:

- Lớp chịu hỗ trợ một object constructor string.



- Lớp mang tính tập kết được (poolable), với giới hạn trên là 100 và kích thước ban đầu là 5.
- Lớp chịu hỗ trợ duy nhất một hàm hành sự có thể thành công hoặc thất bại. Để báo cho Runtime biết tình trạng hiện hành, hàm hành sự này hỗ trợ attribute AutoComplete.

Sau đây là toàn bộ bảng liệt in:

### ***Thí dụ 7-15: Lớp DotNetCOMPlusServer***

```
*****
using System;
using System.EnterpriseServices;
using System.Windows.Forms;

namespace DotNetCOMPlusServer
{
    [ObjectPooling(true, 5, 100)]
    [ConstructionEnabledAttribute(true)]
    [ClassInterface(ClassInterfaceType.AutoDual)]
    public class ComPlusType: ServicedComponent
    {
        public ComPlusType() {}

        // Thi công IObjectConstruct.
        protected override void Construct(string msg)
        {
            MessageBox.Show(msg, "Ctor string is");
        }

        // Thi công các thành viên abstract được kế thừa
        protected override void Activate()
        {
            MessageBox.Show("In activate!");
        }

        protected override void Deactivate()
        {
            MessageBox.Show("In deactivate!");
        }

        protected override bool CanBePooled()
        {
            return true;
        }

        // hàm hành sự duy nhất COM+ aware
        public void DeleteCar(int id)
        {
            MessageBox.Show("Deleting car number " + id.ToString(),
                            "Delete car");
        }
    }
}
*****
```

Lẽ dĩ nhiên, đối tượng này có vẻ đơn giản quá, vì nó chả có làm chi ra hồn ở cấp xí nghiệp (ngoài việc gỡ bỏ một chiếc xe khỏi căn cứ dữ liệu chẳng hạn). Đây chỉ đủ đối với mục tiêu minh họa của chúng ta, là nghiên cứu các điều cơ bản về tương tác .NET/COM+.

Bạn phải tiến hành thêm một bước trước khi cài đặt assembly này vào một ứng dụng COM+. Vì rằng binary này sẽ chui vào GAC, bạn cần xây dựng một strong name đối với assembly **DotNetComPlusServer.dll** này. Như bạn còn nhớ, trình tiện ích **sn.exe** cho phép tạo một strong name đối với assembly này. Trên command line, bạn viết

```
sn -k theKey.snk
```

thì một tập tin \*.snk sẽ được tạo ra. Khi có tập tin này, bạn cho nhậ tu trên tập tin **AssemblyInfo.cs**, dòng lệnh attribute cấp assembly như sau (dựa vào lối tìm về \*.snk) như sau:

```
[assembly: AssemblyKeyFile(@"D:\thien\theKey.snk")]
```

Ngoài ra, bạn cũng muốn đông cứng phiên bản được kết sinh thành:

```
[assembly: AssemblyVersion("1.0.0.0")]
```

Tới đây bạn có thể cho biên dịch dự án.

### 7.10.1.1 Thêm các attribute thiên COM+ cấp assembly

Tới đây coi như .NET assembly của bạn đã sẵn sàng được cài đặt vào COM+ Catalog. Như bạn sẽ thấy, trình tiện ích **regsvcs.exe** sẽ tự động kết sinh một mã nhận diện ứng dụng AppID và tên ứng dụng. Tuy nhiên, nếu bạn muốn khai báo vài khía cạnh đặc biệt của ứng dụng COM+ bạn có thể thêm một vài attribute cấp assembly vào tập tin **AssemblyInfo.cs**:

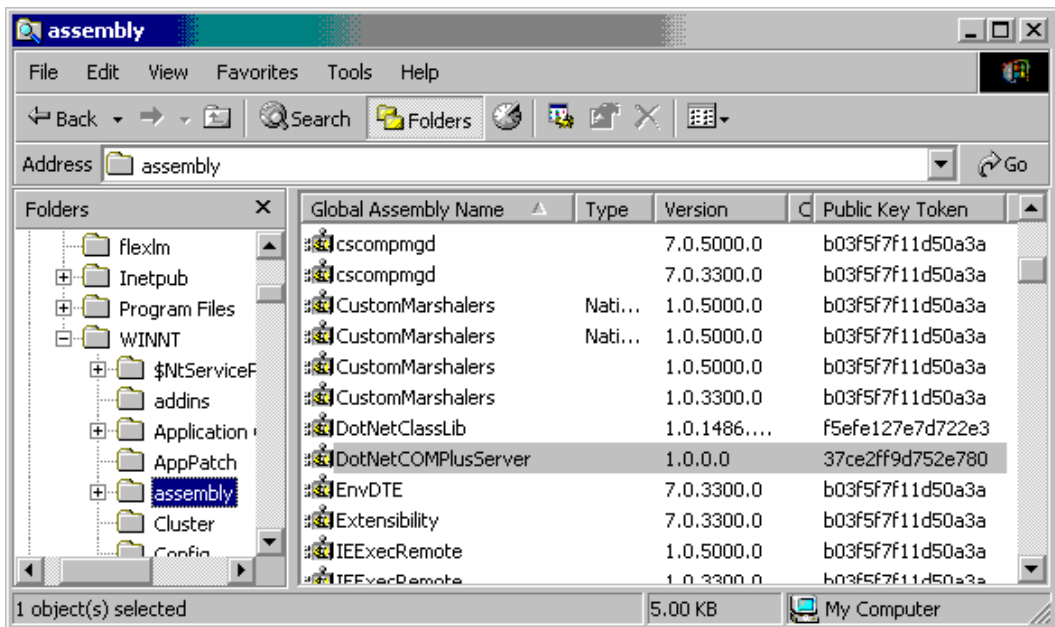
```
[assembly: AssemblyDescription("This app really kicks.")]
[assembly: ApplicationActivation(ActivationOption.Server)]
[assembly: ApplicationID("")]
[assembly: ApplicationName("DotNetComPlusServer")]
```

ApplicationID attribute khỏi phải giải thích. Đây là mã nhận diện ứng dụng COM+ kết xuất. Application Name và Description cũng chả có chi khó hiểu. Chỉ có attribute ApplicationActivation là đáng quan tâm. Bạn nhớ lại ứng dụng MTS và COM+ có thể được lưu trữ như là một library (nghĩa là được kích hoạt trong process của phía triệu gọi) hoặc một server (nghĩa là một thể hiện mới của dllhost.exe). Vì rằng đặt để mặc nhiên dùng cấu hình hóa ứng dụng COM+ là library, do đó bạn muốn khai báo rõ ra là ActivationOption.Server.

### 7.10.1.2 Cấu hình hóa assembly vào COM+ Catalog

Muốn cấu hình hóa một .NET assembly vào COM+ catalog, bạn sẽ phải cho kết sinh một COM type library (sử dụng trình tiện ích **tlbexp.exe**) rồi cho đăng ký kiểu dữ liệu vào system registry (sử dụng trình tiện ích **regasm.exe**). Ngoài ra, bạn còn đưa đúng thông tin vào COM+ catalog (sử dụng RegDB). Thay vì sử dụng các trình tiện ích riêng rẽ này, .NET SDK cung cấp cho bạn một trình tiện ích bổ sung mang tên **regsvcs.exe**. Trình tiện ích này đơn giản hóa tiến trình bằng cách lo tất cả các chi tiết cần thiết trong một bước duy nhất. Đặc biệt các công tác sau đây sẽ được thực hiện:

- Assembly sẽ được nạp vào ký ức.
- Assembly sẽ được đăng ký một cách đúng đắn (nghĩa là giống như với regasm.exe).
- Một type library sẽ được kết sinh và được đăng ký (nghĩa là giống như với tlbexp.exe).
- Type library sẽ được cài đặt trong một ứng dụng COM+ được khai báo.
- Các component sẽ được cấu hình hóa dựa theo những attribute được khai báo trong phân định nghĩa kiểu dữ liệu.



Hình 7-38: Cài đặt các assembly chịu chơi COM+ lên GAC

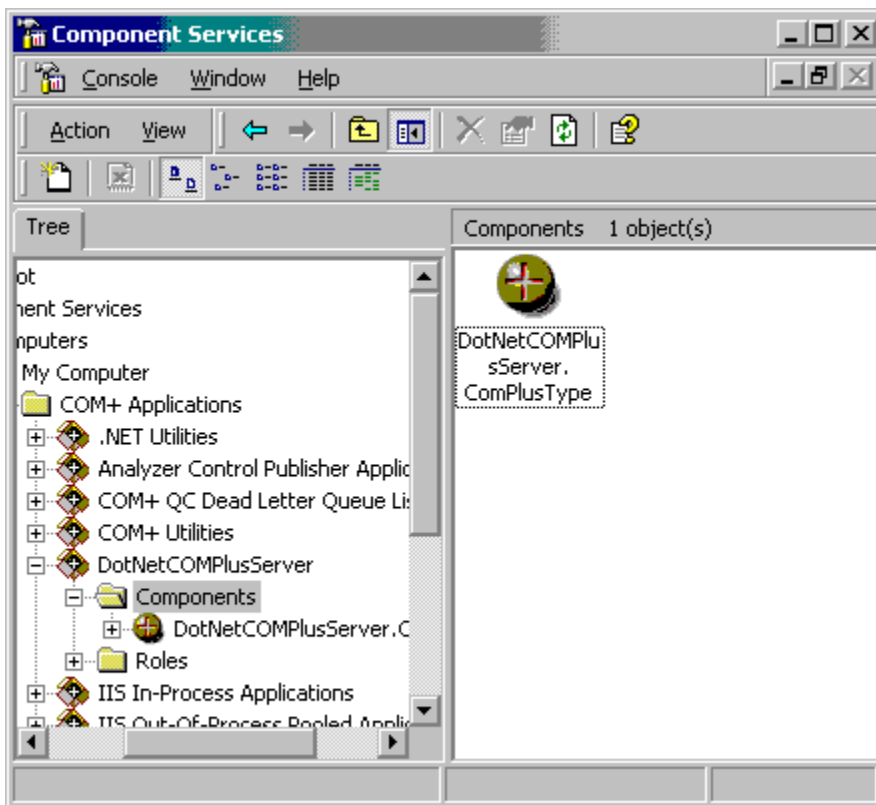
Trong khi công cụ regsvcs.exe này cung cấp một số đối tượng tùy chọn, cú pháp đơn giản nhất sẽ như sau:

```
regsvcs /fc DotNetComPlusServer.dll
```

Ở đây, bạn khai báo flag **/fc** (find hoặc create) báo cho trình tiện ích biết là xây dựng một ứng dụng COM+ nếu hiện chưa hiện hữu một ứng dụng như thế. Bạn có thể tùy chọn khai báo tên của ứng dụng COM+ như là thông số command line. Nếu không ghi ra (như trong trường hợp này) thì tên ứng dụng sẽ dựa trên tên của assembly. Cuối cùng, cho đặt assembly này vào trong GAC (hình 7-38).

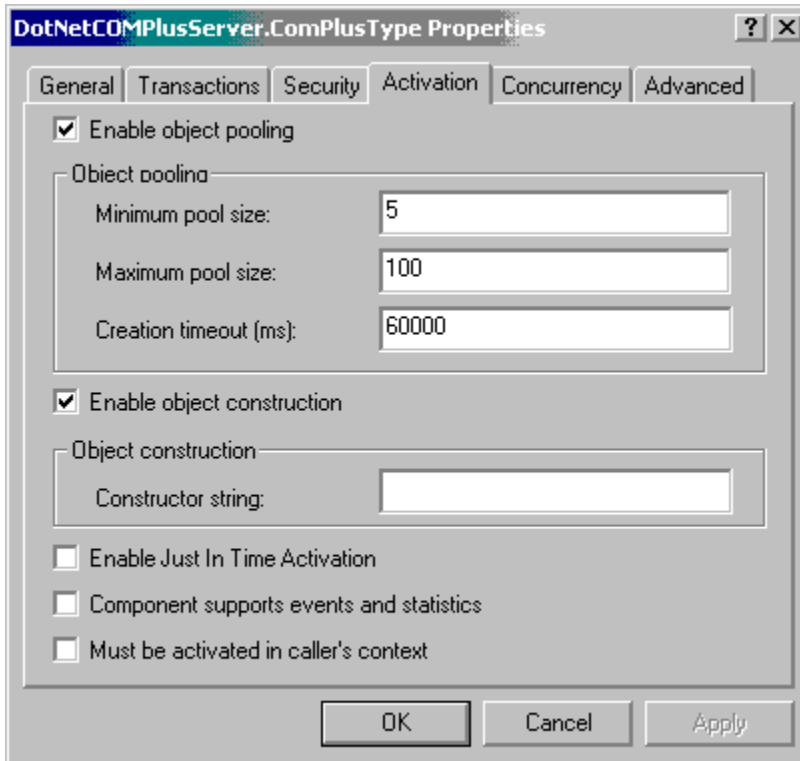
## 7.10.2 Quan sát Component Service Explorer

Một khi bạn thi hành câu lệnh trên bạn có thể mở component này thông qua **Component Services Explorer** của Windows 2000 (bằng cách ra lệnh **Start | Programs | Administrative Tools | Component Service**), và thấy .NET assembly bây giờ là một kiểu dữ liệu COM+ hợp lệ. Xem hình 7-39.



Hình 7-39:: Icon COM+ mang hình viên thuốc aspirine nổi tiếng

Nếu bạn khảo sát các cửa sổ Property khác nhau đối với kiểu dữ liệu COM+ mới này (trên hình 7-39), những thuộc tính khác nhau mà bạn khai báo trên lớp C# sẽ được dùng để cấu hình hóa kiểu dữ liệu của bạn một cách đúng đắn trên COM+ catalog. Thí dụ, nếu bạn right-click lên DotNetComPlusServer, rồi click mục Property trên trình đơn shortcut, rồi chọn tab Activation, hình 7-40, bạn thấy các attribute trên lớp C# đã được ghi lên đây.



**Hình 7-40: Component được cấu hình hoá**

## Chương 8

# Lập trình ứng dụng Web với Web Forms và ASP.NET

Mãi tới nay, phần lớn các thí dụ ứng dụng của bạn hoặc sử dụng console (window desktop) hoặc Windows Forms (ứng dụng client-server). Trong chương này, chúng tôi bắt đầu khảo sát việc viết các ứng dụng Web sử dụng sàn diễn .NET. Để bắt đầu, chúng ta sẽ lược qua khái niệm cơ bản về Web, bao gồm HTML, HTTP request (POST và GET), vai trò của kịch bản phía client (sử dụng JavaScript), và ASP (ActiveX Server Page) cổ điển. Nếu bạn đã rành Web, thì bạn có thể bỏ qua phần này.

Như bạn sẽ thấy, ASP.NET sẽ hỗ trợ một mô hình lập trình khá mạnh so với ASP cổ điển. Thí dụ, giờ đây bạn có thể phân HTML riêng rẽ thành phần logic trình bày (presentation logic) và phần logic business sử dụng một kỹ thuật được gọi là *Codebehind*. Ngoài ra, việc xây dựng các ứng dụng Web sử dụng ASP.NET cho phép bạn sử dụng các ngôn ngữ lập trình “thứ thiệt”, chẳng hạn C# và VB .NET, thay vì các ngôn ngữ kịch bản (scripting language). Nếu bạn khảo sát kỹ ứng dụng Web sử dụng ASP.NET, bạn sẽ học nhiều về kiểu dữ liệu **Page** và **Request** giống như ASP cổ điển, và các thuộc tính **Response**, **Session** và **Application**.

Và cuối cùng, chương này sẽ xem xét vai trò của các ô control phía server (chẳng hạn WebForm Controls), kiểm tra hợp lệ và các tình huống phía server.

Khi nào bạn nắm vững nội dung chương này, bạn sẽ học tiếp trong chương kế tiếp đề mục ASP.NET Web Services.

Web Forms đem lại những kỹ thuật RAD (Rapid Application Development) trong việc triển khai những ứng dụng Web. Giống như với Windows Forms (xem chương 2, “Xây dựng một ứng dụng Windows Forms”), bạn cho lôi thả các ô control Web lên biểu mẫu rồi bổ sung những đoạn mã viết theo kiểu in-line hoặc theo kiểu code-behind page. Tuy nhiên, với Web Forms, ứng dụng sẽ được triển khai trên một web server, và người sử dụng sẽ tương tác với ứng dụng thông qua một “bộ rão xem<sup>25</sup>” (browser).

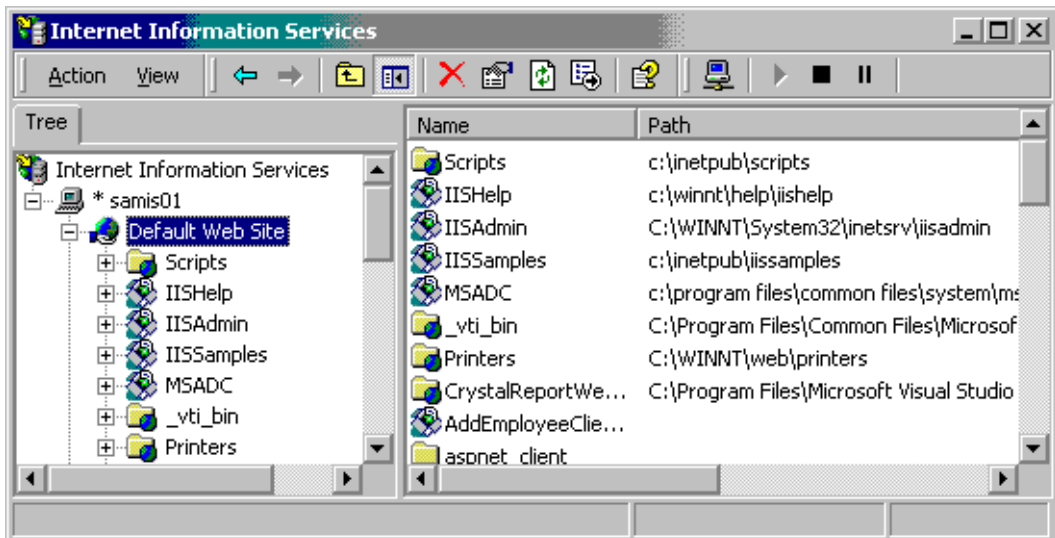
---

<sup>25</sup> Người ta thường gọi là “trình duyệt”. Chúng tôi dịch là “bộ rão xem” nghĩa là cho rão qua một văn bản để xem nội dung.

## 8.1 Phân biệt Web Application và Web Serser

Trước khi đi sâu vào ASP.NET Framework, chúng ta nên dành chút thời gian xem lại kiến trúc cơ bản của một ứng dụng Web đơn giản và những công nghệ cốt lõi thiên về Web trong tiến trình xây dựng một ứng dụng Web. Để bắt đầu, một *ứng dụng Web* có thể được xem như là một tập hợp những tập tin có liên hệ (\*.htm, \*.asp, \*.aspx, các tập tin hình ảnh, v.v..) cũng như những cấu kiện có liên hệ (cấu kiện .NET hoặc cấu kiện COM cổ điển) được trữ trên một Web Server.

Một *Web server* là một sản phẩm phần mềm lo việc lưu trữ (hosting) các ứng dụng của bạn và điển hình cung cấp một số dịch vụ có liên hệ chẳng hạn an toàn hội nhập (integrated security), hỗ trợ FTP (File Transfer Protocol), dịch vụ trao đổi thư điện tử, v.v.. IIS (Internet Information Server) là sản phẩm Web server cấp xí nghiệp của Microsoft. Hiện thời, IIS đã ở phiên bản 6.0 và được xem như là thành phần của hệ điều hành Window 2000.



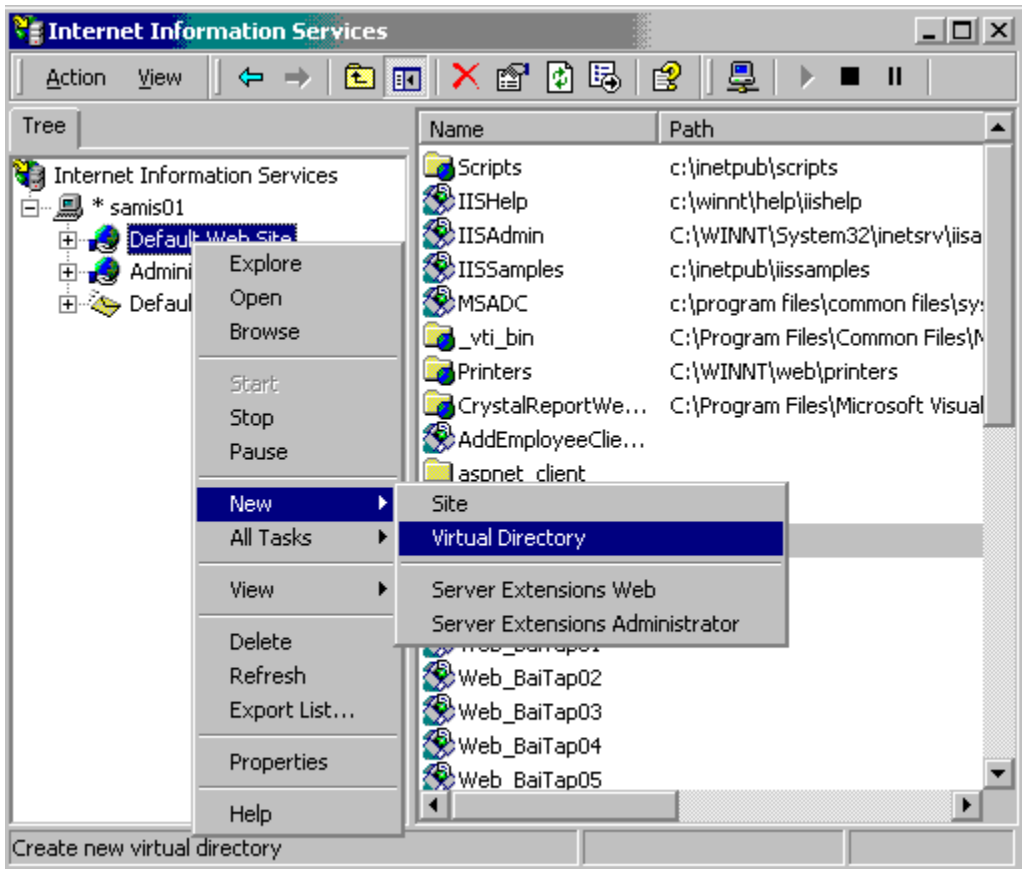
Hình 8-01: IIS applet

Khi bạn tạo các ứng dụng Web với ASP cổ điển hoặc với ASP.NET, bạn sẽ phải trực tiếp hoặc gián tiếp tương tác với IIS. Tuy nhiên, bạn nên nhớ là IIS không tự động được chọn ra khi bạn cài đặt Windows 2000 Professional Edition. Do đó, bạn phải cho cài đặt bằng tay IIS trước khi thử các thí dụ trong chương này. Muốn cài đặt IIS, bạn chỉ cần truy cập Add/Remove Program từ Control Panel và chọn “Add/Remove Windows Components”.

Giả sử bạn đã cài đặt IIS trên máy tính của bạn, thông qua Control Panel, bạn có thể tương tác với IIS từ thư mục Administrative Tools. Đối với chương này, bạn chỉ liên quan đến Default Web Site node (hình 8-01).

## 8.1.1 Tìm hiểu Virtual Directories

Một cài đặt IIS nào đó có khả năng lưu trữ vô số ứng dụng Web, mỗi ứng dụng sẽ tá túc trên một *thư mục ảo* (virtual directory), được ánh xạ lên một thư mục vật lý trên ổ đĩa cục bộ.



**Hình 8-02: Tạo một virtual directory**

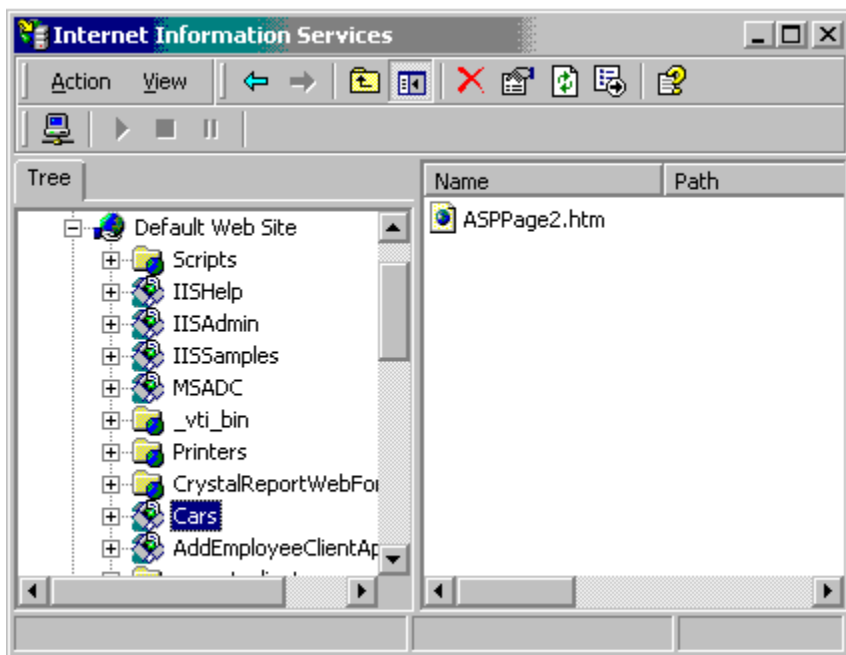
Do đó, khi bạn tạo một ứng dụng Web mang tên FrogsAreUs (ta là ếch ương), thì thiên hạ ở ngoài sẽ lái đến web site của bạn sử dụng đến một URL (Universal Resource Locator), chẳng hạn <http://www.FrogsAreUs.com>. Tuy nhiên, về mặt nội bộ, ứng dụng FrogsAreUs Web có thể lại ánh xạ lên một thư mục vật lý chẳng hạn "C:\FrogsSite" chứa



tập hợp các tập tin thuộc thành phần ứng dụng Web này.

Để minh họa tiến trình, bạn cho tạo một ứng dụng Web đơn giản, cho mang tên **Cars** chẳng hạn. Bước đầu tiên là cho tạo một thư mục mới trên máy tính của bạn dùng trữ tập hợp các tập tin của web site này (chẳng hạn D:\Thien\CarsWebSite). Sau khi tạo xong thư mục, bạn sẽ phải tạo một thư mục ảo mới dùng lưu trữ **Cars** site. Có nhiều cách thực hiện điều này sử dụng IIS applet, một cách đơn giản là right-click lên nút gút Default Web Site (hình 8-01), rồi ra lệnh **New | Virtual Directory** trên trình đơn shortcut. Xem hình 8-02, trang trước.

Lúc này một wizard sẽ được khởi động. Bạn bỏ qua màn hình chào mừng, bạn cho Web site mang một cái tên (Cars). Tiếp theo, wizard yêu cầu bạn cho biết thư mục vật lý trên ổ đĩa chứa những tập tin khác nhau cũng như hình ảnh tượng trưng cho web site của bạn (trong trường hợp này là D:\Thien\CarsWebSite).



**Hình 8-03: Thư mục ảo mới của Bạn: Cars**

Bước cuối cùng trên wizard sẽ yêu cầu bạn cho biết những nét cơ bản liên quan đến thư mục ảo này (chẳng hạn truy cập read/write đối với những tập tin nó chứa, khả năng nhìn xem nội dung các tập tin này trên một Web browser, khả năng khởi động các chương trình khả thi .EXE (chẳng hạn ứng dụng CGI). Trong trường hợp của chúng ta, các chọn lựa mặc nhiên đã quá đủ (Bao giờ bạn cũng có thể thay đổi những chọn lựa sau

khi chạy công cụ này sử dụng cửa sổ Properties). Một khi xong việc với wizard, bạn sẽ thấy là thư mục ảo mới của bạn đã được đăng ký với IIS (hình 8-03).

## 8.2 Tìm hiểu về Web Forms

Web Forms thiết đặt một mô hình lập trình theo đây các trang web sẽ được tạo động trên một web server để chuyển qua cho một browser trên Internet. Trong chừng mực nào đó, các trang web được xem như là kế tiếp của những trang ASP, và chúng kết hợp công nghệ ASP với lập trình truyền thống.

Với Web Forms, bạn tạo ra một trang HTML với nội dung static, và bạn viết ra đoạn mã C# để kết sinh nội dung động. Đoạn mã C# chạy trên server, và dữ liệu được kết xuất sẽ hội nhập vào phần HTML static để tạo ra trang web. Những gì chuyển cho browser là HTML chuẩn.

Web Forms được thiết kế để chạy trên bất cứ browser nào, với server tô vẽ (render) đúng HTML chịu chơi với một browser nào đó. Bạn có thể lập trình phần logic của Web Forms theo bất cứ ngôn ngữ .NET nào. Trong bộ sách này, chúng tôi chọn C#, nhưng một số lập trình viên khác có thể chọn VB.NET.

Giống như với Windows Forms, bạn có thể tạo Web Forms trên Notepad (hoặc trên một trình soạn thảo văn bản bạn chọn) thay vì trên Visual Studio.NET. Nhiều lập trình viên làm như thế, nhưng Visual Studio.NET làm cho tiến trình thiết kế và trắc nghiệm Web Forms dễ dàng hơn nhiều.

Web Forms chia giao diện người sử dụng thành 2 phần: phần nhìn thấy được (còn gọi là phần visual UI - user interface), và phần logic nằm đằng sau. Nó cũng tương tự như triển khai Windows Forms, nhưng với Web Forms thì trang UI và đoạn mã sẽ nằm trên những tập tin riêng rẽ.

Trang UI sẽ được trữ trên một tập tin mang phần nối rộng \*.aspx. Đoạn mã logic đối với trang web này có thể được trữ trong một tập tin nguồn *code-behind* C# riêng biệt. Khi bạn cho chạy biểu mẫu, tập tin lớp code-behind sẽ chạy và tạo động HTML để chuyển cho browser. Đoạn mã này sử dụng các lớp Web Forms nằm trong hai namespace **System.Web** và **System.Web.UI**.

Với Visual Studio.NET, lập trình Web Forms không gì dễ dàng bằng bạn cho mở một trang web, cho lôi một vài ô control web thả lên trang, rồi viết đoạn mã lo thụ lý các tình huống. Thế là xong, bạn đã viết một ứng dụng Web.

Mặt khác, cho dù với Visual Studio.NET việc viết một ứng dụng web mạnh “cứng

cấp” và trọn vẹn cũng là một công việc đầy nhọc nhằn. Web Forms cung cấp một UI rất phong phú; số ô control web với độ phức tạp được phát triển ngày càng nhiều trong những năm gần đây và người sử dụng chờ đợi sự hấp dẫn của các ứng dụng Web ngày càng tăng.

Ngoài ra, các ứng dụng Web mang tính phân tán (distributed). Điển hình là khách hàng sẽ không nằm trong cùng tòa nhà với server. Đối với phần lớn các ứng dụng web bạn phải để ý đến băng thông (bandwidth) cũng như hiệu năng của mạng khi tạo UI. Một hành trình từ client về host có thể mất vài giây.

## 8.2.1 Các tình huống trên Web Form

Web Forms mang tính “vận hành theo tình huống” (event-driven). Một *tình huống* (event) là một đối tượng gói ghém ý tưởng “là cái gì đó sẽ xảy ra”. Một tình huống sẽ được kết sinh (raised, nổi lên) khi người sử dụng ấn một button hoặc chọn ra một mục trên ô list control hoặc nói cách khác tương tác với khung giao diện UI. Các tình huống cũng có thể được phát sinh bởi hệ thống khi khởi động hoặc kết thúc công việc. Thí dụ, bạn cho mở đọc một tập tin, và hệ thống sẽ phát ra một tình huống khi tập tin được đọc ghi vào ký ức.

Hàm hành sự thụ lý tình huống được gọi là hàm thụ lý tình huống (event handler), được viết theo C# trên trang code-behind và được gắn liền với những ô control trên trang HTML thông qua những control attribute.

Các hàm thụ lý tình huống là những delegate. Theo qui ước, các hàm thụ lý tình huống ASP.NET trả về void và tiếp nhận hai thông số. Thông số đầu tiên tìm thấy cho đối tượng đã phát ra tình huống (gọi là sender), còn thông số thứ hai được gọi là *đối mục tình huống* (event argument) chứa những thông tin đặc thù của tình huống. Đối với phần lớn các tình huống, event argument đều thuộc kiểu dữ liệu **EventArgs** không trưng ra bất cứ thuộc tính nào. Đối với vài ô control, event argument có thể được dẫn xuất từ **EventArgs** và có thể trưng ra những thuộc tính đặc thù đối với loại tình huống này.

Trong các ứng dụng web, phần lớn các tình huống đều được thụ lý trên server và như vậy đòi hỏi một chuyến đi vòng vo. ASP.NET chỉ hỗ trợ một số hạn chế tình huống, chẳng hạn button click và text change. Đây là những tình huống mà người sử dụng có thể chờ đợi gây ra một thay đổi đáng kể, ngược lại với những tình huống Windows, chẳng hạn mouse-over, có thể xảy ra nhiều lần trong một công tác duy nhất.

### 8.2.1.1 Các tình huống postback so với non-postback

Các *tình huống postback*<sup>26</sup> là những tình huống gây ra việc biểu mẫu được gọi trả về cho server ngay lập tức. Các tình huống loại này bao gồm click type event, chẳng hạn tình huống Button Click. Ngược lại, nhiều tình huống (điển hình là change event) được xem như là *non-postback event* theo đây biểu mẫu không được gọi trả về cho server ngay lập tức. Thay vào đó, các tình huống này được “che cất” (cached) bởi ô control cho tới khi một postback event xảy ra. Bạn cũng có thể ép những ô control mang tính non-postback event phải hành động theo kiểu postback event bằng cách cho đặt thuộc tính **AutoPostBack** về true.

### 8.2.1.2 Tình trạng châu làm việc

*State* của một ứng dụng web là trị hiện hành của tất cả các ô control và biến đổi với người sử dụng hiện hành trong châu làm việc hiện hành. Web vốn dĩ là một môi trường vô tình trạng (stateless). Đây có nghĩa là mỗi lần post vào server sẽ làm mất đi tình trạng của những lần post đi trước, trừ phi lập trình viên phải mất công gìn giữ tình trạng của châu làm việc. Tuy nhiên, ASP.NET cung cấp việc hỗ trợ sự duy trì gìn giữ tình trạng của một châu làm việc của người sử dụng.

Bất cứ lúc nào người sử dụng yết thị (post) lên server, họ sẽ làm lại từ đầu trước khi được trả về cho browser. ASP.NET cung cấp một cơ chế tự động duy trì tình trạng đối với mỗi ô server control. Do đó, nếu bạn cung cấp một danh sách và người sử dụng chịu chọn ra, thì sự lựa chọn sẽ được bảo tồn gìn giữ sau khi trang web được yết thị vào lại server và được tô vẽ lại lên phía client.

## 8.2.2 Chu kỳ sống của Web Form

Mỗi yêu cầu cung cấp một trang web từ phía web server sẽ gây ra một dây chuyền các tình huống ở phía server. Những tình huống này, từ đầu đến cuối, hình thành một *chu kỳ sống* (life cycle) đối với trang web cũng như đối với tất cả các thành viên trang web. Chu kỳ sống bắt đầu từ yêu cầu cung cấp một trang web, làm cho server phải nạp trang này vào. Khi yêu cầu đã được thỏa mãn, trang sẽ được tổng khứ (unloaded). Từ đầu chu kỳ đến cuối chu kỳ, mục tiêu là cho hiện lên kết xuất trang tài liệu HTML thích ứng lên browser đã phát ra yêu cầu. Chu kỳ sống của một trang web được đánh dấu bởi những tình huống sau đây, mỗi tình huống sẽ được thụ lý bởi bạn hoặc để cho mặc nhiên thụ lý bởi ASP.NET server:

---

<sup>26</sup> Từ “post” ở đây có nghĩa là “yết thị” trước bàn dân thiên hạ (ở đây là browser) công bố cho biết gì đó.

### ***Initialize***

Initialize (khởi gán) là giai đoạn đầu tiên trong chu kỳ sống đối với bất cứ trang web hoặc ô control nào. Trong giai đoạn này bạn cho khởi gán bất cứ những đặt để nào trong suốt thời gian của yêu cầu đi vào.

### ***Load View State***

Thuộc tính **ViewState** của ô control sẽ được điền đầy dữ liệu. Thông tin **ViewState** được lấy từ một biến ẩn dấu trên ô control, biến này dùng giữ lại **State** suốt cuộc hành trình về server. Chuỗi input được trích từ biến hidden này sẽ được parsed bởi page framework, và thuộc tính **ViewState** sẽ được đặt để. Điều này có thể bị thay đổi thông qua hàm hành sự **LoadViewState()**, cho phép ASP.NET quản lý tình trạng của ô control xuyên qua việc nạp các trang web, như vậy mỗi ô control sẽ không bị reset về tình trạng mặc nhiên của nó mỗi lần trang web được posted.

### ***Process Postback Data***

Trong giai đoạn này, dữ liệu được chuyển cho server trong tiến trình yết thị sẽ được xử lý. Nếu bất cứ dữ liệu nào đưa đến một yêu cầu nhật tu **ViewState** thì việc nhật tu sẽ được thực hiện thông qua hàm hành sự **LoadPostData()**.

### ***Load***

Hàm hành sự **CreateChildControls()** sẽ được triệu gọi, nếu thấy cần thiết, để tạo và khởi gán các ô control server trong control tree. **State** được hoàn nguyên, và các ô form control sẽ cho thấy dữ liệu phía client. Bạn có thể thay đổi giai đoạn nạp này bằng cách thụ lý tình huống **Load** thông qua hàm hành sự **OnLoad()**.

### ***Send Postback Change Modifications***

Nếu có bất cứ thay đổi nào giữa tình trạng hiện hành và tình trạng đi trước, thì tình huống **Change** sẽ được phát ra thông qua hàm hành sự **RaisePostDataChangeEvent()**.

### ***Handle Postback Events***

Tình huống phía client gây ra postback sẽ được thụ lý.

### ***PreRender***

Đây là giai đoạn ngay trước khi kết xuất được vẽ lên browser. Đây thực chất là cơ may cuối cùng đối với bạn để thay đổi kết xuất trước khi tô vẽ lên browser, bằng cách dùng hàm hành sự **OnPreRender()**.

### ***SaveState***

Vào gần đầu chu kỳ sống, view state được nạp vào từ một biến cất giấu. Bây giờ nó được cất trữ trở lại lên biến này, dưới dạng một đối tượng kiểu chuỗi hoàn tất hành

trình về client. Bạn có thể phủ quyết điều này bằng cách dùng hàm hành sự **SaveViewState()**.

### **Render**

Chính ở đây là nơi kết xuất được gửi trả về cho browser phía client sẽ được kết sinh. Bạn có thể phủ quyết việc này bằng cách sử dụng hàm hành sự **Render**. Nếu thấy cần thiết, hàm hành sự **CreateChildControl()** sẽ được triệu gọi vào để tạo và khởi gán những ô server control trên control tree.

### **Dispose**

Đây là giai đoạn chót của chu kỳ sống. Nó cho bạn cơ may thực hiện bất cứ việc dọn dẹp nào và giải phóng những qui chiếu đối với bất cứ nguồn lực mắc mớ nào, chẳng hạn kết nối căn cứ dữ liệu. Bạn có thể thay đổi nó bằng cách dùng hàm hành sự **Dispose()**.

## **8.3 Cấu trúc cơ bản của một tài liệu HTML**

Xem như bạn có sẵn một thư mục ảo, bạn cần tạo bản thân ứng dụng Web. Khi tạo các ứng dụng Web, bạn không thể nào tránh việc sử dụng ngôn ngữ HTML (HyperText Markup Language). Như có thể bạn đã biết, HTML là ngôn ngữ “đánh dấu” (markup) chuẩn được dùng mô tả làm thế nào các văn bản, hình ảnh, kết nối (link) cũng các ô control HTML GUI được thể hiện bởi Web browser. Trong khi các IDE hiện đại (bao gồm Visual Studio.NET) có vô số công cụ “bẩm sinh” che lấp HTML thô thiển khỏi mắt mọi người, bạn vẫn phải cảm thấy thoải mái với HTML khi làm việc với ASP.NET.

Một tập tin HTML thường bao gồm một tập hợp cốt lõi những HTML tag.<sup>27</sup> Các tag này được dùng khai báo cho biết là một tập tin HTML, thông tin tổng quát về tài liệu (tiêu đề, metadata tập tin, v.v..) cũng như phần thân tài liệu (nghĩa là tập hợp văn bản, hình ảnh, bảng dữ liệu, link, v.v..). Bạn nên nhớ tag không phân biệt chữ hoa hoặc chữ thường (ta gọi là case-insensitive). Do đó, đối với browser, <HTML>, <html> và <Html> đều giống nhau.

Để bắt đầu, bạn cho mở IDE của Visual Studio.NET, rồi đưa vào một tập tin HTML trống rỗng bằng cách ra lệnh **Files | New | Files** rồi cho cất trữ tập tin này trên thư mục mặc nhiên như là default.htm. Nếu bạn quan sát tập tin \*.htm mới được tạo ra bởi IDE bạn sẽ thấy khung sườn (skeleton) như sau:

```
<HTML>
<HEAD>
<TITLE></TITLE>
```

---

<sup>27</sup> Tag, tạm dịch là cái “đỉnh bài” trên áo các quan lại cho biết chức tước.

```
<META NAME="GENERATOR" Content="Microsoft Visual Studio">
<META HTTP-EQUIV="Content-Type" content="text.html">
</HEAD>
<BODY>

<!-- Chèn HTML ở đây-->

</BODY>
</HTML>
```

Một tag X HTML nào đó sẽ được mở với ký hiệu `<X>` và được đóng lại bởi ký hiệu `</X>`. Cặp tag `<HTML>` và `</HTML>` dùng đánh dấu điểm khởi đầu và điểm cuối tài liệu HTML. Như bạn có thể đoán ra được là Web browser sẽ dựa trên các tag này để biết bắt đầu từ đâu sẽ hiển thị dạng thức được khai báo trong thân tài liệu.

Các tag `<HEAD>` dùng trữ bất cứ metadata nào liên quan đến bản thân tài liệu. Ở đây, HTML header dùng vài tag `<META>` để mô tả xuất xứ của tập tin này (MS Visual Studio) và nội dung tập tin. Hiện thời trang Web của ta không có tựa đề (title), do đó ta thử thay đổi thành như sau:

```
<HTML>

<HEAD>
<TITLE> Khó tránh khỏi HTML </TITLE>
<META NAME="GENERATOR" Content="Microsoft Visual Studio">
<META HTTP-EQUIV="Content-Type" content="text.html">
</HEAD>

<BODY>

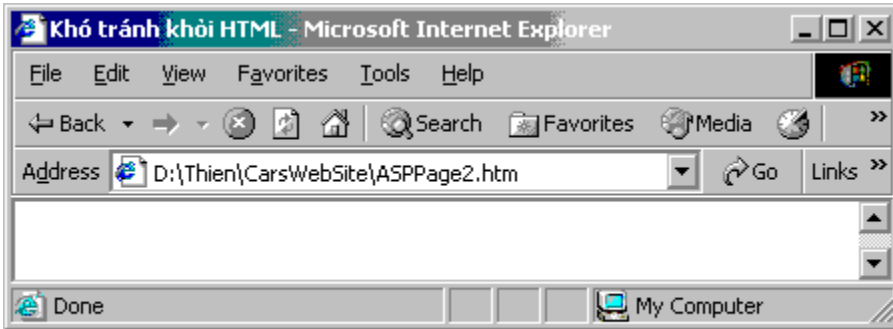
<!-- Chèn HTML ở đây-->

</BODY>

</HTML>
```

Tag `<TITLE>` dùng khai báo chuỗi văn bản sẽ được ghi ra lên trên thanh tựa đề (title bar) của hosting Web browser. Một khi bạn cho cất trữ tập tin này và mở lại trên một browser, bạn sẽ thấy một cái gì đó giống như hình 8-04. Bạn để ý tựa đề cửa sổ “Khó tránh khỏi HTML”.

Hành động thật sự đằng sau một tập tin HTML sẽ diễn ra trong cặp tag `<BODY>`. Nằm trong lòng cặp `<BODY>` `</BODY>` là bất cứ số lượng nào các tag bổ sung được dùng để thể hiện và định dạng thông tin văn bản cũng như đồ họa. Tập sách này không đi sâu vào từng tag HTML. Tuy nhiên, trong những trang tài liệu kế tiếp một vài tag cốt lõi sẽ được trình bày khi làm việc với những ứng dụng ASP.NET.



Hình 8-04: Tag &lt;TITLE&gt; vào cuộc

### 8.3.1 Cơ bản về định dạng văn bản trên HTML

Sử dụng khá rõ ràng của một tập tin HTML là cho hiển thị những thông điệp dạng văn bản. Trên trang HTML, các phần tử văn bản thường được đưa vào trong lòng tag <BODY>. Thí dụ, bạn muốn tạo một trang theo dõi việc đăng nhập (login) đối với một ứng dụng Web. Bạn để ý cú pháp chú giải trên HTML: <!-- ->

```
<BODY>
  <!-- Câu nhắc tưởng đối với người sử dụng ->
  Trang đăng nhập Cars
</BODY>
```

Bạn để ý là trong trường hợp này, bạn không cần bao bởi tag tương ứng. Khi browser gặp phải một dòng văn bản không có tag thì nó tuân in ra thông tin như đã viết. Như vậy, khi bạn nhậ tu <BODY> thêm một dòng văn bản (in đậm như sau:

```
<BODY>
  <!-- Câu nhắc tưởng đối với người sử dụng ->
  Trang đăng nhập Cars
  Yêu cầu khô vào user name và password
</BODY>
```

thì browser sẽ không thêm một line break như ta mong đợi, xem hình 8-05:

Muốn văn bản ghi ra theo nhiều hàng, bạn cần thêm tag <P> và </P> (tắt chữ Paragraph), yêu cầu browser bắt đầu một paragraph mới.

```
<BODY>
  <!-- Câu nhắc tưởng đối với người sử dụng ->
  Trang đăng nhập Cars
  <p> Yêu cầu khô vào user name và password. </p>
</BODY>
```





Hình 8-05: Thông tin văn bản không có tag sẽ không xuống hàng

Hình 8-06 cho thấy kết xuất.



Hình 8-06: Tag <P> cho bắt đầu một paragraph mới

Muốn đưa vào một hàng trắng (thay vì một paragraph mới), bạn dùng tag <br> (tất chữ “break”, sang hàng) như sau:

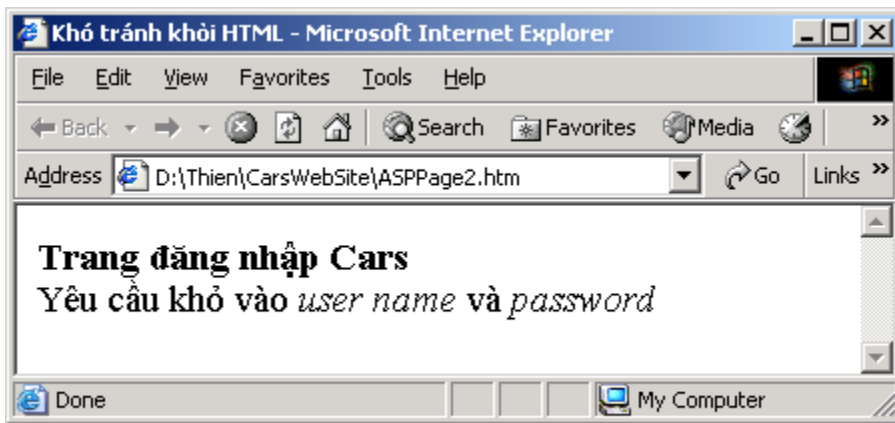
```
<BODY>
  <!-- Cho sang hàng -->
  Trang đăng nhập Cars
  <br> Yêu cầu khỏ vào user name và password. </br>
</BODY>
```

Kết quả giống như hình 8-06 nhưng khoảng cách giữa các hàng ngắn hơn. Tag <br> bắt đầu một hàng mới.

Bây giờ, bạn có thể thêm nhiều hàng văn bản (với sang hàng, carriage return, CR) có thể bạn muốn định dạng các dòng văn bản bằng cách cho in đậm (bold) hoặc in nghiêng (italic), sử dụng đến các tag **<B>** (tắt chữ Bold, in đậm) và **<I>** (tắt chữ Italic, in nghiêng). Thí dụ, bạn cho in đậm hàng đầu tiên và cho in nghiêng một số từ trên hàng thứ hai, như sau:

```
<BODY>
  <!-- Cho in đậm và nghiêng -->
  <b>Trang đăng nhập Cars</b>
  <br> Yêu cầu khỏ vào <i>user name</i> và <i>password</i>. </br>
</BODY>
```

Hình 8-07 cho thấy kết xuất:



Hình 8-07: Cho in đậm và nghiêng

## 8.3.2 Làm việc với Format Headers

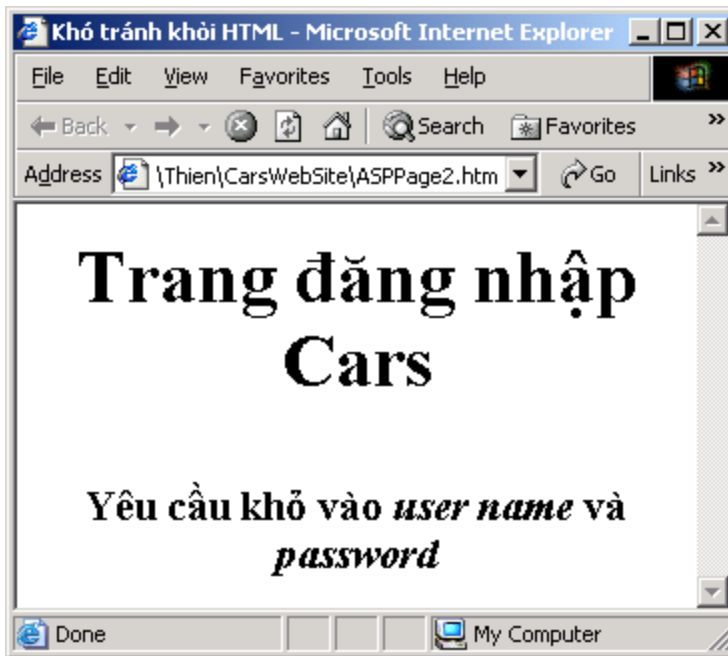
Bây giờ ta xét đến việc sử dụng những tag liên quan đến tiêu đề (header) khác nhau. Bằng cách sử dụng các tag **<h1>**, **<h2>**, **<h3>**, **<h4>**, **<h5>**, và **<h6>** bạn có thể thay đổi cách hiển thị khổ cao của đoạn văn bản. **<h1>** là khổ cao nhất, còn **<h6>** là khổ nhỏ nhất. Sau đây là thí dụ:

```
<BODY>
  <!-- Câu nhắc nhở -->
  <h1>Trang đăng nhập Cars</h1>
  <br> <h3> Yêu cầu khỏ vào <i>user name</i> và
                                <i>password</i>.</h3> </br>
</BODY>
```

Cuối cùng, bạn có thể dùng tag <center> (cũng như left, right hoặc justify) để ép một khối văn bản canh giữa trên vùng client area của browser, như sau:

```
<BODY>
  <!-- Câu nhắc nhở -->
  <center>
    <h1>Trang đăng nhập Cars</h1>
    <br> <h3> Yêu cầu khỏ vào <i>user name</i> và
                                   <i>password</i>.</h3> </br>
  </center>
</BODY>
```

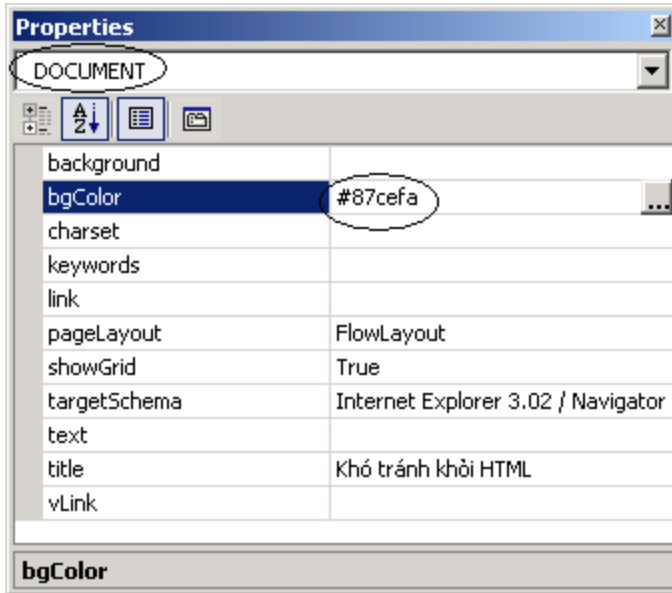
Hình 8-08 cho thấy kết quả cuối cùng. Bạn để ý là khi bạn chỉnh lại kích thước browser, thì văn bản tiếp tục canh giữa.



Hình 8-08: Làm việc với Header Tag

### 8.3.3 Visual Studio.NET HTML Editors

Tới đây, trang HTML của bạn chưa có chỉ là hấp dẫn. Bây giờ, ta thử xem qua vài công cụ thiết kế mà Visual Studio.NET cung cấp cho bạn. Bạn có thể cấu hình hóa những khía cạnh khác nhau của trang HTML bằng cách dùng cửa sổ **Properties**. Muốn thế, bạn chọn ra đối tượng DOCUMENT (hình 8-09). Thí dụ, nếu bạn thay đổi thuộc tính bgColor (background color, màu nền), thì tập tin HTML sẽ tự động được nhật tu. Xem hình 8-10.



Hình 8-09: Hiệu định visual một tài liệu HTML

quả những thay đổi khi thiết kế.

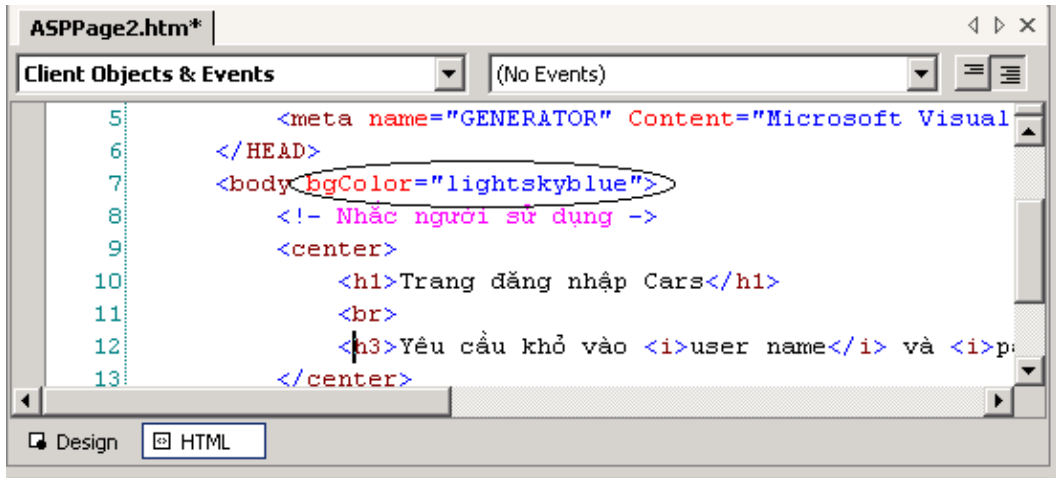
Tới đây, coi như bạn đã biết cách bố trí ban đầu của một trang HTML cơ bản. Bây giờ, bạn muốn biết qua cách làm thế nào tiếp nhận sự tương tác với người sử dụng.

## 8.4 Triển khai HTML Form

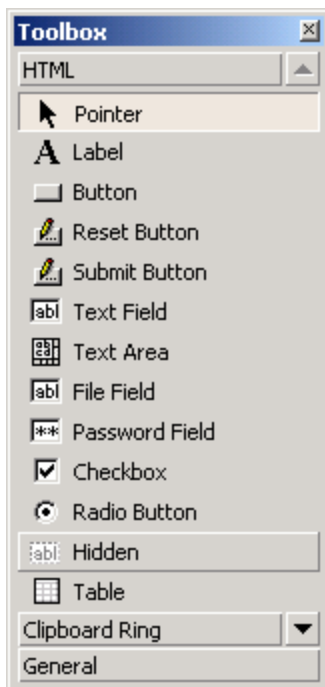
Bạn sẽ thấy về sau trong chương này, ASP.NET Framework sẽ cung cấp một số ô WebForm control chịu trách nhiệm kết sinh tự động những tag HTML. Thông qua các WebForm control này, bạn có thể tạo ra những giao diện người sử dụng UI của trang được trả về không cần biết đến HTML nằm đằng sau. Những ô control mà chúng ta sẽ xét đến trong chốc lát *không phải* là .NET WebForm control, mà đơn giản chỉ là tập hợp những built-in widget<sup>28</sup> được sử dụng trong lúc triển khai HTML Forms.

IDE còn cung cấp một thanh công cụ HTML **Formatting** cho phép bạn thay đổi dáng dấp phần văn bản của bạn (color, font, header size, bullet point, v.v.). Muốn cho xuất hiện thanh công cụ này, bạn ra lệnh **View | Toolbars | Formatting**, thì thanh công cụ sẽ hiện lên trên đầu màn hình. Bạn có thể xây dựng trang HTML theo kiểu Word chẳng hạn. Bạn có thể thử các kiểu định dạng khác nhau rồi xem kết quả xuất hiện các tag thế nào. Trên màn hình Visual Studio.NET IDE, bạn chỉ cần click qua lại nút Design và HTML để thấy kết

<sup>28</sup> Widget, tắt chữ window gadget, là những đồ “lục lãng lục chớt” của Windows.



Hình 8-10: Các thay đổi vào lúc thiết kế được ghi nhận như là HTML



Hình 8-11: Các ô control HTML

Một HTML Form đơn giản chỉ là một nhóm (có mang tên) các phần tử UI có liên hệ với nhau được dùng thu thập dữ liệu do người sử dụng đưa vào để rồi được chuyển cho ứng dụng Web thông qua HTTP. Bạn chớ nhầm lẫn một biểu mẫu HTML với vùng client area được hiển thị bởi browser. Trong thực tế, một HTML form thường còn hơn nhiều là một nhóm logic các widget được đặt nằm trong cặp tag `<form>` và `</form>` như sau:

```

<form name = MainForm id = MainForm>
    <!--Thêm các phần tử UI ở đây-->
</form>

```

Ở đây, bạn đã tạo ra một biểu mẫu, gán cho nó một mã nhận diện ID và một tên thân thiện là MainForm. Việc làm này không bắt buộc, tuy nhiên bạn cũng nên tập làm quen làm việc này. Về sau trong chương này, bạn sẽ thấy là điều hữu ích khi làm việc với kịch bản phía client, theo đấy bạn thường xuyên cần nhận diện các ô control dựa theo tên.

Điển hình, tag `<form>` mở đầu cung cấp một thuộc tính hành động, cho biết URL cần dùng đến để trình duyệt dữ liệu biểu mẫu (submit form data), kể cả hàm hành sự chuyển đi dữ liệu (posting hoặc getting). Bạn sẽ xét đến

trong chốc lát khía cạnh của tag `<form>`. Tạm thời chúng ta thử xem đến loại item mà ta có thể đưa vào một HTML form. Visual Studio.NET IDE cung cấp một hộp đồ nghề

HTML toolbox cho phép bạn chọn ra mỗi HTML-based UI widget. Xem hình 8-11, trang trước.

Bảng 8-01 cho biết ý nghĩa của các item thông dụng của toolbox này.

**Bảng 8-1: Các kiểu HTML GUI thông dụng**

HTML GUI Widget	Mô tả
<b>Button</b>	Một nút không hỗ trợ type attribute được dùng để kích hoạt (trigger) một SUBMIT hoặc RESET. Loại nút này có thể được dùng để thúc một khối mã kịch bản phía client hoặc bất cứ đoạn mã nào không cần một vòng qua Web server.
<b>Checkbox</b> <b>Radio Button</b> <b>Listbox</b> <b>Dropdown</b>	Các ô control UI chuẩn.
<b>Image</b>	Cho phép bạn khai báo một hình ảnh cần được hiển thị trên biểu mẫu.
<b>Reset Button</b>	Nút này có mang type attribute cho về RESET, yêu cầu browser cho xoá trắng các trị trong mỗi ô control trên trang HTML và cho về trị mặc nhiên.
<b>Submit Button</b>	Nút này có mang type attribute cho về SUBMIT, cho chuyển đi dữ liệu biểu mẫu về nơi yêu cầu.
<b>Text Field</b> <b>Text Area</b> <b>Password Field</b>	Ô control này dùng trữ một hàng đơn (hoặc nhiều hàng) văn bản. Password Field sẽ hiển thị dữ liệu nhập sử dụng dấu hoa thị (*) làm character mask.

Bạn nên nhớ các lớp cơ bản .NET cung cấp cho bạn một số kiểu dữ liệu managed tương ứng với những HTML widget thô thiên này. Bạn nên tìm xem namespace **System.Web.UI.HtmlControls**.

## 8.4.1 Xây dựng User Interface

Bước đầu tiên trong việc tạo một giao diện người sử dụng UI sử dụng các HTML widget là khai báo một phân đoạn <form> trên tài liệu HTML. Thí dụ ta thêm markup sau đây:

```
<HTML>
  <HEAD>
    <title>Khó tránh khỏi HTML </title>
    <meta name="GENERATOR" Content="Microsoft Visual Studio.NET 7.0">
    <meta http-equiv="Content-Type" content="text/html">
  </HEAD>

  <body bgColor="lightskyblue">
```





Hình 8-12: Trang Web với khung hình ảnh (rất tiếc không có)

## 8.4.2 Thêm vào một hình ảnh

Mục cuối cùng là làm thế nào đưa hình ảnh vào tài liệu HTML. Giống như các khía cạnh khác trên tài liệu HTML, các hình ảnh cũng sẽ được đánh dấu bởi tag `<img>` (tắt chữ image) như sau đây:

```

```

Thuộc tính **alt** (tắt chữ alternative) dùng khai báo một đoạn văn bản tương đương với hình ảnh đồ họa được khai báo bởi thuộc tính **src** (tắt chữ source). Đoạn văn bản này sẽ hiện lên khi con nháy chuột đang nằm trên chỗ hình ảnh sẽ hiện lên hoặc đối với những browser không hỗ trợ hình ảnh thì đây là văn bản thay thế. Còn thuộc tính **border** là tùy chọn dùng cho hiện lên một cái khung bao quanh hình ảnh.

Bạn nhớ cho trị được gán cho thuộc tính **alt** có thể là một lối tìm về ghi chặt vào (hard-coded path), như trong trường hợp ở đây, hoặc không khai báo một path. Cách tiếp



cận này giả sử là hình ảnh được sử dụng sẽ nằm cùng thư mục như với các tập tin \*.htm sử dụng chúng. Hình 8-12 cho thấy trang Web được nhậ tu hình ảnh (rất tiếc là chúng tôi không có hình ảnh chiếc xe!).



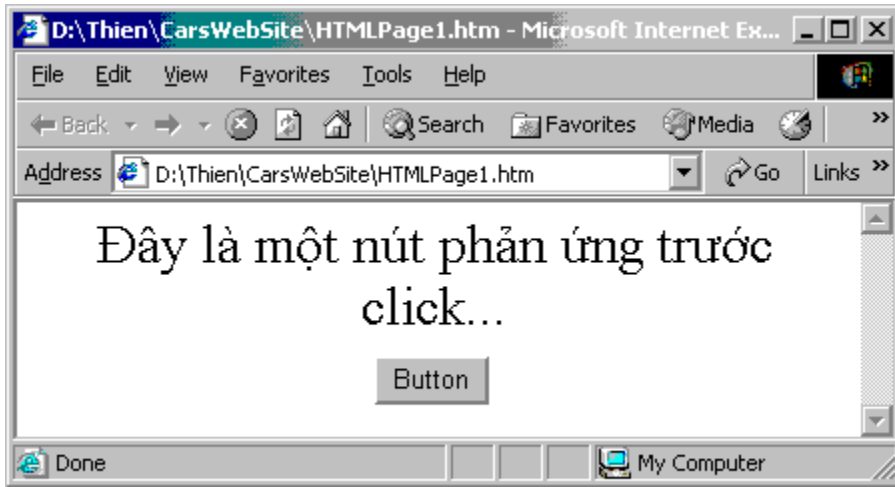
Hình 8-12: Trang Web với khung hình ảnh (rất tiếc không có)

## 8.5 Vai trò kịch bản phía Client

Tới đây bạn đã hiểu thấu cách xây dựng một biểu mẫu HTML. Bước kế tiếp là xem xét đến vai trò của kịch bản phía client. Khía cạnh tôi tậ trong một ứng dụng Web là thường xuyên cần triệu gọi hàm máy phía sever để nhậ tu HTML được hiển thị trên browser. Lẽ dĩ nhiên là không tránh khỏi những chuyển đi đi về về như thế, nhưng bao giờ phải cũng phải quan tâm đến việc giảm thiểu việc di chuyển trên mạng. Một kỹ thuật giúp giảm thiểu việc di chuyển là sử dụng kịch bản phía client để kiểm tra hợp lệ (validation) dữ liệu nhập trước khi submit dữ liệu biểu mẫu về chỗ chứa (recipient).

Thí dụ, hiện thời ta yêu cầu người sử dụng khỏ vào user name và password. Nếu cả hai vùng mục tin này là trắng, bạn sẽ không được phép submit dữ liệu của biểu mẫu. Lẽ

dĩ nhiên, là HTML không thể giúp ích gì trong trường hợp này, vì HTML chỉ có nhiệm vụ là cho hiển thị nội dung trang Web mà thôi. Để thêm chức năng cho HTML chuẩn, bạn phải sử dụng đến một ngôn ngữ kịch bản nào đó (hoặc nếu cần có thể là bất cứ số ngôn ngữ kịch bản nào đó).



Hình 8-13: Một trang HTML mới

Có nhiều loại ngôn ngữ kịch bản. Hai ngôn ngữ phổ biến nhất là **VBScript** và **JavaScript**. VBScript là một tập hợp con của ngôn ngữ lập trình Visual Basic 6.0. Bạn nên nhớ là Microsoft Internet Explorer (IE) là Web browser duy nhất có những hỗ trợ được cài sẵn chịu hỗ trợ VBScript phía client. Do đó, nếu bạn muốn các trang HTML của bạn chạy tốt trên bất cứ Web browser nào có mặt trên thị trường, thì bạn *không* nên dùng VBScript làm logic kịch bản phía client. Thật ra, thì VBScript đã thực sự “ngủ” khi .NET ra đời. Lý do rất đơn giản là khác với ASP, ASP.NET không sử dụng ngôn ngữ kịch bản. Đúng ra, các trang ASP.NET dùng toàn ngôn ngữ lập trình đầy đủ lễ bộ (chẳng hạn VB.NET, C#, v.v..) để thực hiện phần logic phía server.

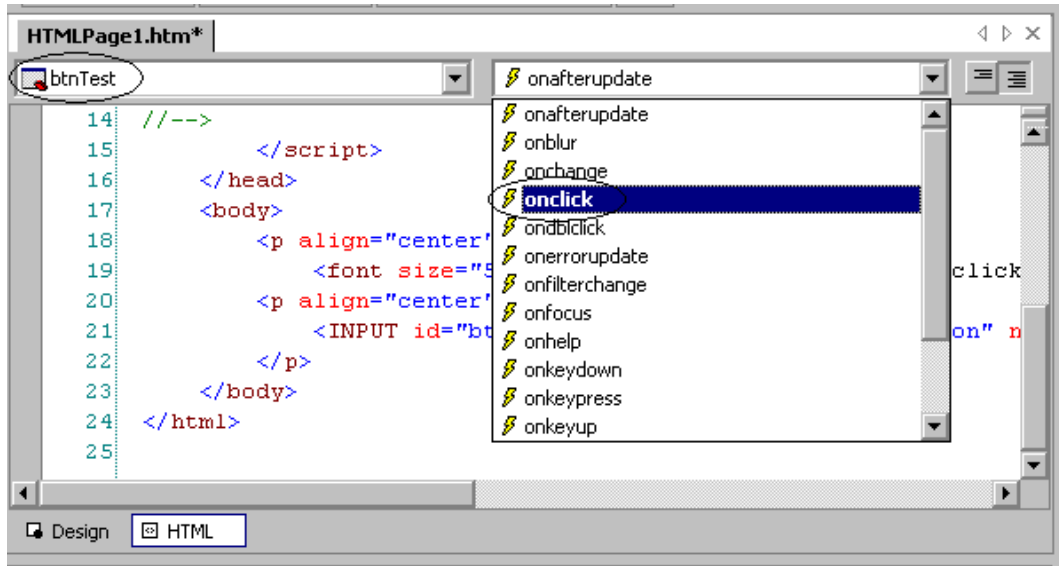
Ngôn ngữ kịch bản thông dụng kia là **JavaScript**. Mặc dù mang tiền tố “Java”, và có cú pháp tương tự với Java, nhưng JavaScript không dính dáng chi với ngôn ngữ lập trình Java, và cũng không phải là một ngôn ngữ lập trình đầy đủ lễ bộ nên không mạnh bằng Java. Điểm son đáng nhắc tới là tất cả các Web browser đều chịu hỗ trợ JavaScript, nên xem JavaScript như là một ứng viên tự nhiên đối với việc kiểm tra hợp lệ phía client.

### 8.5.1 Một thí dụ về Client-Side Scripting

Muốn hiểu sâu về kịch bản phía client, trước tiên bạn cần xét đến làm thế nào chặn hững các tình huống phát ra từ các ô control trên trang HTML. Giả sử bạn có một trang

HTML rất đơn giản như theo hình 8-13.

Tiếp theo, bạn cần gán một mã nhận diện ID và tên button hợp lệ sử dụng cửa sổ Properties (btnTest). Muốn chặn hứng tình huống click đối với button này, bạn cho hiện dịch HTML view và chọn button từ ô liệt kê kéo xuống bên trái. (btnTest) bạn dùng ô liệt kê kéo xuống bên phải chọn ra tình huống onclick (hình 8-14).



Hình 8-14: Chặn hứng các tình huống của HTML widget

Một khi bạn làm xong các điều kể trên, bạn sẽ thấy HTML nhật tu như sau, theo các phần in đậm:

```
<html>
<head>
<title></title>
<meta name="GENERATOR" content="Microsoft Visual Studio.NET 7.0">
<meta name="vs_targetSchema"
    content="http://schemas.microsoft.com/intellisense/ie5">

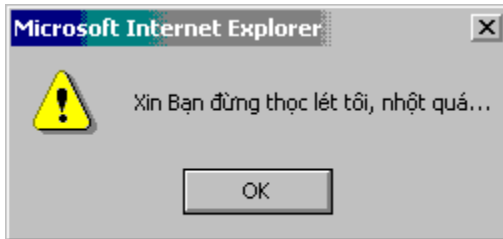
<script id=clientEventHandlersJS language=javascript>
<!--
function btnTest_onclick() {
}
//-->
</script>
</head>
<body>
<p align="center">
<font size="5">Đây là một nút phản ứng trước click...</font></p>
```

```

<p align="center">
<INPUT id="btnTest" type="button" value="Button" name="Button"
  language=javascript onclick="return btnTest_onclick()">
</p>
</body>
</html>

```

Như bạn có thể thấy, một khối `<script>` mới được thêm vào tiêu đề của HTML, với JavaScript là ngôn ngữ kịch bản được chọn và với chú giải kiểu HTML. Lý do rất đơn giản: nếu trang HTML của bạn chạy trên một browser không hỗ trợ JavaScript thì đoạn mã sẽ được xem như là chú giải và bị phớt lờ. Lẽ dĩ nhiên, trang HTML có thể mang ít chức năng nhưng mặt kia trang HTML sẽ không bị “vỡ tung” khi được hiển thị lên browser.



**Hình 8-15: IE Alert**

Kế tiếp, bạn thấy bộ thuộc tính của nút `<Button>` có một thành viên mới mang tên `onclick`, được gán cho hàm JavaScript mới. Như vậy, khi nút `<Button>` bị click thì hàm này sẽ tự động được triệu gọi. Để cho đơn giản, bạn cho nhậ tu hàm này như sau:

```

<script id=clientEventHandlersJS language=javascript>
<!--
function btnTest_onclick()
{
    // Cho hiển thị thông điệp
    alert("Xin bạn đừng thọc lét tôi, nhột quá...");
}
//-->
</script>

```

Và bạn sẽ thấy một message box hiện lên khi bạn ấn nút `<Button>`. Xem hình 8-15.

## 8.5.2 Kiểm tra hợp lệ đối với trang HTML Page1.htm

Bây giờ, bạn cho nhậ tu trang HTMLPage1.htm cho hỗ trợ kiểm tra hợp lệ phía client. Mục tiêu là bảo đảm khi người sử dụng click nút `<Summit>` bạn sẽ cho triệu gọi một hàm JavaScript kiểm tra text box không được trống rỗng. Nếu đúng là trường hợp này, thì cho phát ra một báo động (alert) yêu cầu người sử dụng khò lại dữ liệu. Trước tiên, gán một tình huống `onclick` cho nút `<Submit>` về một hàm hành sự JavaScript mang tên `ValidateData()`. Trong hàm này bạn viết các lệnh kiểm tra các text box như sau:





báo method = “GET” như là chế độ truyền tin, thì dữ liệu biểu mẫu sẽ được ghi nối đuôi vào một query string (chuỗi truy vấn) như là một tập hợp những cặp name/value. Hàm hành sự khác về truyền tin dữ liệu biểu mẫu về Web server sẽ được khai báo bởi method = “POST” như sau:

```
<form name=MainForm
action=http://localhost/Cars/ClassicASPPage.asp method = "POST">
.
.
.
</form>
```

Trong trường hợp này, dữ liệu biểu mẫu sẽ không được ghi nối đuôi vào query string, mà thay vào đó là được viết ra trên một hàng riêng biệt với tiêu đề HTTP. Theo thể thức này, dữ liệu biểu mẫu không trực tiếp nhìn thấy được đối với thế giới bên ngoài và như vậy sẽ an toàn hơn. Trong tạm thời, giả sử bạn khai báo GET method trong việc chuyển dữ liệu.

## 8.6.1 Phân tích ngữ nghĩa của một Query String

Muốn hiểu chính xác làm thế nào tập tin tiếp nhận ASP trích ra dữ liệu biểu mẫu, bạn cần quan sát query string. Khi bạn submit dữ liệu biểu mẫu sử dụng đến GET action, bạn sẽ thấy một chuỗi văn bản xuất hiện trên ô Address của browser. Sau đây là một thí dụ:

```
http://localhost/Cars/ClassicASPPage.asp?
txtUserName=Duong&txtPassword=somepassword&btnSubmit=Submit
```

Một chức năng cốt lõi của query string này là dấu chấm hỏi (?) phân cách. Bên trái dấu ? là địa chỉ nơi tiếp nhận (trang ASP của bạn) Còn bên phải dấu ? là một chuỗi gồm bởi bất cứ cặp name/value (chẳng hạn txtUserName=Duong).

Như bạn có thể thấy mỗi cặp name/value được phân cách bởi dấu ampersand (&). Query string này khá đơn giản để phân tích ngữ nghĩa vì rằng bạn không đưa vào khoảng trắng trong tiến trình. Tuy nhiên, khi user name bị thay đổi từ Duong qua Duong Quang, thì bạn sẽ thấy query string như sau:

```
http://localhost/Cars/ClassiASPPage.asp?
txtUserName=Duong+Quang&txtPassword=somepassword&btnSubmit=Submit
```

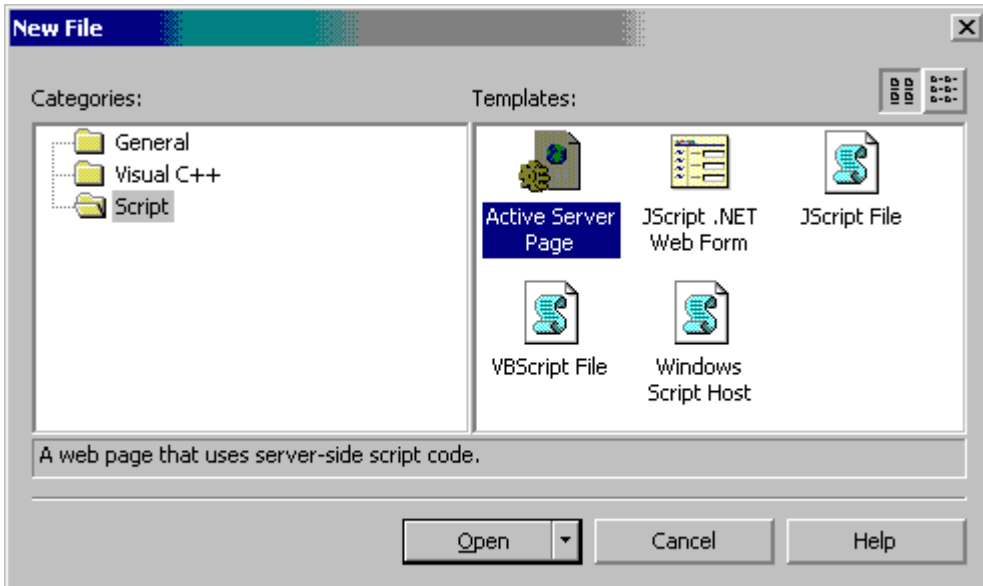
Bạn để ý mỗi ký tự trắng được thay thế bởi một dấu cộng (+). Như vậy, nếu giữa Duong và Quang có 5 ký tự trắng, thì query string sẽ lại như sau:

```
http://localhost/Cars/ClassiASPPage.asp?
txtUserName=Duong+++++Quang&txtPassword=somepassword&btnSubmit=Submit
```

Ngoài việc thụ lý các ký tự trắng, query string còn tượng trưng những ký tự khác nhau đặc biệt (chẳng hạn ^ và ~) như dưới dạng hexa tương ứng. Do đó, nếu bạn submit lại trang ASP sử dụng Hello^^77 như là password, bạn sẽ thấy như sau:

```
http://localhost/Cars/ClassiASPPage.asp?txtUserName=
Duong+++++Quang&txtPassword=Hello%5E77&btnSubmit=Submit
```

## 8.7 Tạo một Active Server Page cổ điển



Hình 8-16: Chèn vào một tập tin ASP cổ điển

Muốn tiếp nhận dữ liệu biểu mẫu, bạn cần tạo một tập tin ClassicASPPage.asp, rồi đưa vào một tập tin Active Server Page mới sử dụng Visual Studio.NET (hình 8-16), bằng cách ra lệnh **New | File** rồi chọn **Script** trên khung Categories và **Active Server Page** trên khung Template. Bạn phải chắc chắn là tên tập tin bạn gán cho item này là cùng tên như được khai báo trên action attribute của biểu mẫu và bảo đảm là tập tin này cũng được trữ trên cùng thư mục ảo được ánh xạ.

Một ASP là một đoạn mã kịch bản HTML “hỗ lớn” phía server. Nếu bạn chưa hề làm việc với ASP cổ điển thì bạn hiểu cho mục đích của ASP là tạo ra một HTML động vào lúc chạy (on the fly) sử dụng kịch bản phía server. Thí dụ, có thể bạn có một khối kịch bản đọc một bảng dữ liệu từ dữ liệu nguồn (sử dụng ADO) và trả về các hàng dữ liệu như là HTML generic.

Đối với thí dụ này, trang ASP sẽ dùng đối tượng bả sinh ASP **Request** để đọc những trị của query string đi vào rồi cho hiển thị như là HTML (như vậy là phản hồi - echoing - nhập liệu) Sau đây là kịch bản có ý nghĩa (bạn để ý việc sử dụng `<%...%>` để đánh dấu một khối kịch bản:



```

<%@ Language=VBScript %>
<!-- VBScript 8-OK trên phía server -->
<html>
<head>
<meta name="GENERATOR" Content="Microsoft Visual Studio.NET 7.0">
</head>
<body>
<!-- Trả lui info chúng gọi cho ta -->
<center>
    <h1> bạn bảo:</h1>
    <b>User Name:</b><%=Request.QueryString("txtUserName") %><br>
    <b>Password:</b><%=Request.QueryString("txtPassword") %><br>
</center>
</body>
</html>

```

Điều đầu tiên phải để ý là một tập tin \*.asp bắt đầu và kết thúc với những cặp tag <html>, <head>, và <body>. Ở đây bạn dùng đối tượng **Request**, giống như bất cứ kiểu dữ liệu COM cổ điển nào chịu hỗ trợ một số thuộc tính, hàm hành sự và tính hướng. Bạn triệu gọi hàm **QueryString()** để xem xét các trị chứa trong mỗi HTML widget (được submit thông qua “method=GET”). Ngoài ra, bạn cũng ghi nhận là ký hiệu <%=...%> là cách viết tắt bảo rằng “Cho chèn phần sau đây vào trả lời HTTP”. Muốn uyển chuyển hơn, bạn có thể dùng trực tiếp đối tượng ASP **Response**. Sau đây là một thí dụ:

```

<!-- Trả lui info chúng gọi cho ta -->
<center>
    <h1> bạn bảo:</h1>
    <b>User Name:</b><%=Request.QueryString("txtUserName") %><br>
    <b>Password:</b>
    <%
        dim pwd
        pwd = Request.QueryString("txtPassword")
        Response.Write (pwd)
    %><br>
</center>

```

Các đối tượng **Request** và **Response** của ASP cổ điển cung cấp thêm một số thành viên. Ngoài ra, lớp ASP cũng định nghĩa một số nhỏ đối tượng (**Session**, **Server**, **Application**, và **ObjectContext**) mà bạn có thể dùng trong khi tạo ứng dụng Web. Ở đây bạn không có thời gian xem xét các đối tượng này. Tuy nhiên, vào cuối chương bạn sẽ thấy cùng cách hành xử sẽ được cung cấp bằng cách dùng các thuộc tính của kiểu dữ liệu ASP.NET Page.

Trong bất cứ trường hợp nào, muốn kích hoạt logic ASP, bạn chỉ cần khởi động trang default.htm từ một browser rồi cho submit thông tin. Sau khi kích bản được xử lý, bạn trở về tập tin HTML mới được kết sinh động. Xem hình 8-17:

Coi như xong. Thí dụ hiện hành không có chi hấp dẫn lắm. Tuy nhiên, bạn đã hiểu các nguyên tắc chủ chốt nằm đằng sau lập trình ASP (và ASP.NET). Dựa trên dữ liệu được submit bởi một biểu mẫu HTML, bạn có thể dùng đoạn mã để trả về động nội dung cho người sử dụng.



Hình 8-17: HTML được kết sinh động

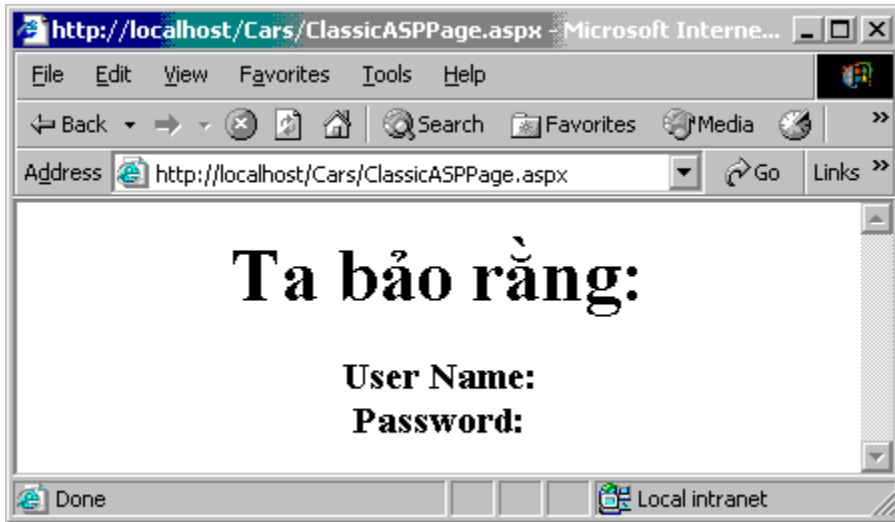
## 8.7.1 Đáp lại POST Submissions

Hiện hành tập tin default.htm cho biết GET là hàm hành sự chuyển đi dữ liệu biểu mẫu về trang tiếp nhận \*.asp. Với cách tiếp cận này, các trị được chứa trong những GUI widget khác nhau sẽ được ghi nối đuôi lên vào cuối query string. Điểm quan trọng phải nhớ cho là hàm hành sự **Request.QueryString()** của ASP chỉ có khả năng trích dữ liệu được submit thông qua hàm hành sự GET. Nếu bạn thay đổi hàm hành sự chuyển dữ liệu thành action=POST và trở về ứng dụng Web, bạn sẽ nhận trả lời trống rỗng. Xem hình 8-18.

Đây là vì dữ liệu biểu mẫu bây giờ được gọi đi như là thành phần của HTTP header, thay vì là thông tin văn bản ghi nối đuôi. Điểm đáng mừng là cũng thông tin này có thể nhận được bằng cách sử dụng collection **Request.Form**. Muốn submit dữ liệu sử dụng POST, bạn có thể nhậ tu tập tin \*.asp như sau:

```
<! Trả lui info chúng gọi cho ta ->
<center>
  <h1> bạn bảo:</h1>
  <b>User Name:</b><%=Request.Form("txtUserName") %><br>
  <b>Password:</b>
  <%
```

```
dim pwd
pwd = Request.Form("txtPassword")
Response.Write(pwd)
%><br>
</center>
```



Hình 8-18: Hàm `QueryString()` chỉ xử lý thông tin submit bởi GET

Một khi làm xong việc này, bạn có thể đọc vào dữ liệu. Tuy nhiên, lần này các trị không được ghi nối đuôi về URL (hình 8-19).

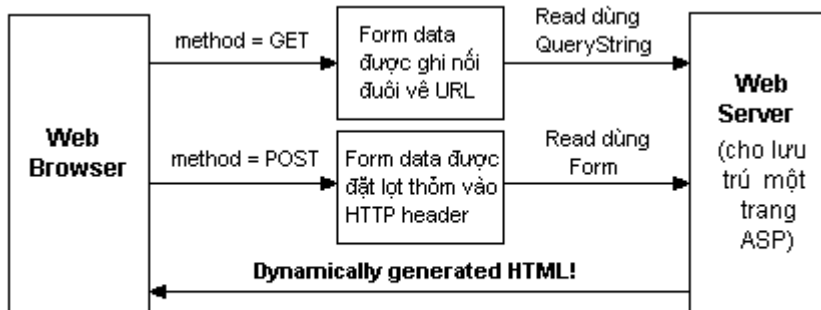


Hình 8-19: Dữ liệu được POST phải sử dụng `Request.Form()`

Hình 8-20 minh họa các kỹ thuật được sử dụng để submit dữ liệu biểu mẫu đối với một recipient và kỹ thuật tương ứng để nhận dữ liệu biểu mẫu từ một ứng dụng ASP Web cổ điển.



Hình 8-19: Dữ liệu được POST phải sử dụng Request.Form()



Hình 8-20: Submitting dữ liệu cho một trang ASP sử dụng HTTP GET và POST

## 8.8 Xây dựng một ứng dụng ASP.NET

Trước khi kết thúc việc duyệt xem các điều cơ bản về Web, bạn cho mở default.htm và nhập tu tag <form> mở đầu như sau (bạn để ý phần đuôi \*.aspx).

```
<form name=MainForm
action=http://localhost/Cars/ClassicASPPage.aspx method=post ID=Form1>
```

Bạn cho đổi phần đuôi tập tin ClassicASPPage thành \*.aspx rồi cho chạy lại ứng dụng. Bạn không thấy khác biệt gì cả. Bạn xem hình 8-19, chúng tôi đã thay thế cái đuôi. Coi như bạn đã tạo ứng dụng ASP.NET đầu tiên.

Như bạn có thể thấy, tất cả các kỹ thuật được trình bày mãi tới đây đều hợp lệ trong thế giới ASP.NET. Phần còn lại của chương này là nghiên cứu framework của ASP.NET.

## 8.9 Vài vấn đề đối với ASP cổ điển

Mặc dù có khá nhiều ứng dụng Web thành công trong việc sử dụng ASP cổ điển, nhưng kiến trúc cũng mang nhiều bất cập. Có thể nói bất cập lớn nhất của ASP cổ điển là các ngôn ngữ kịch bản VBScript và JavaScript chẳng hạn. Các ngôn ngữ này thuộc diện diễn dịch (interpreted) nên không thích hợp với kỹ thuật lập trình thiên đối tượng.

Một vấn đề khác đối với ASP cổ điển là trong thực tế một trang \*.asp không phải là một đoạn mã được đơn thể hóa (modularized). Phần lớn các ứng dụng Web là một trộn lẫn rối rắm hai kỹ thuật lập trình khác nhau. Trong một thế giới lý tưởng, một Web framework phải cho phép phân presentation logic (nghĩa là đoạn mã HTML) được tách rời khỏi phần business logic (nghĩa là đoạn mã chức năng).

Vấn đề cuối cùng là việc ASP cổ điển đòi hỏi xử lý tốt phần “đồ ăn nấu sẵn” (boilerplate), kịch bản trùng lặp có chiều hướng lặp lại giữa các dự án. Đa số các ứng dụng Web đều đòi hỏi kiểm tra hợp lệ dữ liệu nhập, hiển thị nội dung phong phú HTML, v.v.. Trên ASP cổ điển, bạn phải lo phần hỗ trợ việc thêm các đoạn mã kịch bản thích ứng phía server. Lý tưởng mà nói, thì Web Framework phải lo các chi tiết này, thay vì để cho lập trình viên “hụp lặn”.

### 8.9.1 Một số lợi điểm của ASP.NET

ASP.NET giải quyết những giới hạn của ASP cổ điển vừa nêu trên. Trước tiên, các tập tin ASP.NET (\*.aspx) không sử dụng ngôn ngữ kịch bản, cho phép bạn sử dụng ngôn ngữ lập trình “thứ thiệt” như C#, VB .NET, hoặc JScript.NET chẳng hạn. Do đó, bạn có thể áp dụng những kỹ thuật bạn đã học qua trong các tập sách này thẳng vào việc triển khai ứng dụng Web. Như bạn có thể chờ đợi, các trang \*.aspx có thể dùng các lớp thư viện .NET cũng như truy cập các chức năng cung cấp bởi các assembly “cây nhà lá vườn”.

Kế tiếp, các ứng dụng .NET đem lại cho bạn nhiều cách giảm thiểu số lượng đoạn mã bạn cần phải viết ra. Thí dụ, thông qua việc sử dụng các Web control, bạn có thể xây dựng một ứng dụng án đầu (front-end app) thiên browser sử dụng những ô web control khác nhau cho ra HTML thô ở hậu trường. Các Web control khác được dùng để tự động kiểm tra hợp lệ các GUI item của bạn (như vậy giảm tối đa kịch bản phía client mà bạn chịu trách nhiệm phải viết).

Ngoài việc đơn giản hóa công sức viết các đoạn mã, ASP.NET còn cung cấp vô số “lục lãng lục chôt” (bell & whistle) rất tiện lợi. Thí dụ, tất cả các ứng dụng Web ASP.NET đều dùng Visual Studio.NET IDE (một cái tiến rất lớn trong việc gỡ rối logic kịch bản sử dụng Visual Interdev).

Bây giờ, ta thử quan sát các namespace cốt lõi của ASP.NET.

## 8.10 ASP.NET Namespaces

Thư viện lớp .NET chứa vô số namespace tượng trưng cho các kỹ thuật thiên về Web. Nhìn chung, các namespace này có thể phân thành 3 nhóm chính: core Web atoms (nghĩa là HTTP type, configuration type, và security type), UI (Web controls) và Web services (sẽ được mô tả ở chương sau). Phải trợn một tập sách mới có thể quan sát mỗi mục liệt kê ở đây. Trong tạm thời, bạn có thể lướt qua các chức năng cung cấp bởi các namespace cốt lõi được mô tả trong bảng 8-2 sau đây:

**Bảng 8-02: Các namespace ASP.NET**

Namespace Web	Mô tả
<b>System.Web</b>	<b>System.Web</b> định nghĩa các kiểu dữ liệu cốt lõi cho phép liên lạc giữa browser/Web (chẳng hạn khả năng <b>Request</b> và <b>Response</b> , cookie manipulation, và chuyển tập tin).
<b>System.Web.Caching</b>	Namespace này chứa những kiểu dữ liệu chịu hỗ trợ caching đối với một ứng dụng Web.
<b>System.Web.Configuration</b>	Namespace này chứa những kiểu dữ liệu chịu hỗ trợ việc cấu hình hóa ứng dụng Web phối hợp với tập tin config Web của dự án.
<b>System.Web.Security</b>	Namespace này chịu hỗ trợ an ninh đối với một ứng dụng Web.
<b>System.Web.Services</b> <b>System.Web.Services.Description</b> <b>System.Web.Services.Discovery</b> <b>System.Web.Services.Protocols</b>	Các namespace này cung cấp những kiểu dữ liệu cho phép bạn xây dựng những dịch vụ Web (sẽ được đề cập đến ở chương 9, “Lập trình Web Service”).
<b>System.Web.UI</b> <b>System.Web.UI.WebControls</b> <b>System.Web.UI.HtmlControls</b>	Các namespace này định nghĩa một số kiểu dữ liệu cho phép bạn xây dựng một GUI front end đối với ứng dụng Web.

### 8.10.1 Các lớp cốt lõi của System.Web

Namespace **System.Web** định nghĩa tập hợp tối thiểu và trọn vẹn các kiểu dữ liệu cho phép một browser-based client liên lạc và tương tác với Web server. Bảng 8-3 liệt kê một vài kiểu dữ liệu đáng quan tâm mà chúng tôi sẽ xét đến chi tiết trong chương này.

**Bảng 8-3: Các kiểu dữ liệu cốt lõi của namespace System.Web**

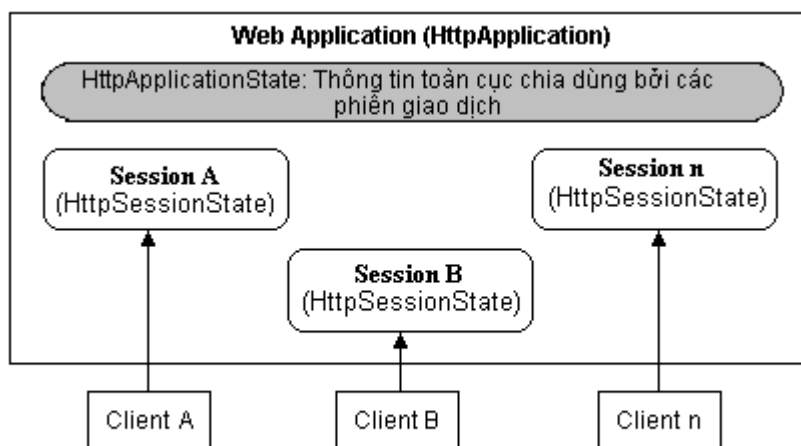
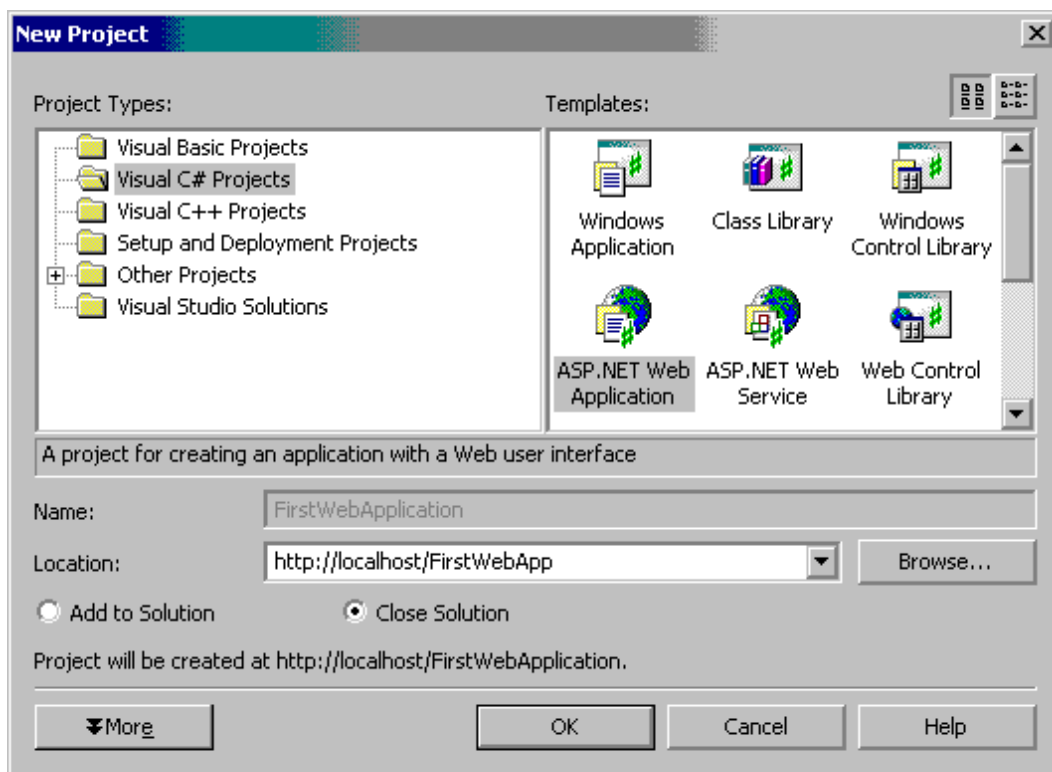
Kiểu dữ liệu System.Web	Mô tả
<b>HttpApplication</b>	Lớp này định nghĩa các thành viên thông dụng đối với tất cả các ứng dụng ASP.NET. Như bạn có thể thấy, tập tin <b>global.asax</b> định nghĩa một lớp được dẫn xuất từ <b>HttpApplication</b> .
<b>HttpApplicationState</b>	Lớp này cho các nhà triển khai khả năng chia sẻ thông tin toàn cục giữa nhiều request, session và pipeline trong một ứng dụng ASP.NET.
<b>HttpBrowserCapabilities</b>	Lớp này cho phép server biên dịch thông tin dựa trên khả năng của browser chạy trên máy tính client.
<b>HttpCookies</b>	Lớp này cung cấp một cách truy cập nhiều HTTP cookies một cách an toàn về kiểu dữ liệu (type safe).
<b>HttpRequest</b>	Lớp này cung cấp một thể thức thiên đối tượng cho phép liên lạc giữa server với browser (nghĩa là dùng truy cập vào dữ liệu HTTP request cung cấp bởi client).
<b>HttpResponse</b>	Lớp này cung cấp một thể thức thiên đối tượng cho phép liên lạc giữa server với browser (nghĩa là dùng chuyển kết xuất về cho một client).

## 8.11 Tìm hiểu sự phân biệt Application/Session

Một khía cạnh mới của lập trình Web đối với lập trình viên ứng dụng máy tính để bàn là sự phân biệt tình trạng giữa ứng dụng và châu giao dịch (session). Như đã được định nghĩa, một ứng dụng Web có thể xem như là một tập hợp các tập tin có liên hệ với nhau nằm trên một thư mục ảo. ASP.NET cung cấp kiểu dữ liệu **HttpApplication** để tượng trưng cho những hàm hành sự, thuộc tính, và tình huống thông dụng đối với một ứng dụng Web nào đó. Như bạn có thể thấy tập tin **globals.asax** định nghĩa một kiểu dữ liệu duy nhất (mang tên **Global**) được dẫn xuất từ lớp cơ bản trừu tượng **HttpApplication**.

Gắn liền với **HttpApplication** là lớp **HttpApplicationState**, cho phép bạn chia sẻ sử dụng các thông tin toàn cục giữa nhiều session trong một ứng dụng ASP.NET. Một châu giao dịch (session) cho biết sự tương tác của người sử dụng với một ứng dụng Web. Thí dụ, nếu 20.000 người sử dụng được đăng nhập truy cập **Cars** site, thì sẽ có 20.000 châu giao dịch đang được tiến hành.

Trên ASP.NET, mỗi châu giao dịch sẽ giữ lại thông tin tình trạng đối với một người sử dụng nào đó, và được tượng trưng về mặt chương trình bởi kiểu dữ liệu **HttpSessionState**. Theo thể thức này, mỗi người sử dụng sẽ được cấp phát một khối ký ức tượng trưng cho việc tương tác duy nhất của người sử dụng đối với ứng dụng Web. Hình 8-21 cho thấy mối liên hệ giữa một ứng dụng Web và những phiên giao dịch Web:

**Hình 8-21: Tình trạng Application và Session****Hình 8-22: Tạo ứng dụng ASP.NET Web đầu tiên**



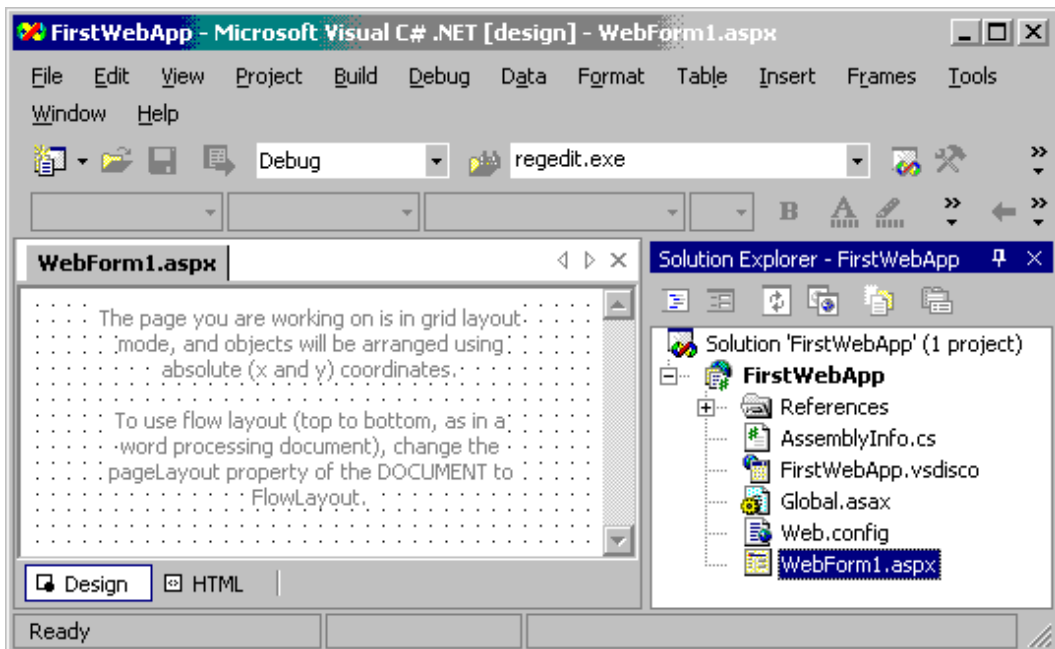
Dưới thời ASP cổ điển, các khái niệm về tình trạng ứng dụng và châu giao dịch được tượng trưng bằng cách dùng những đối tượng riêng biệt (Application và Session). Còn theo ASP.NET, thì một kiểu dữ liệu được dẫn xuất từ Page sẽ định nghĩa giống nhau các thuộc tính mang tên (named Application và Session), cho trưng ra các kiểu dữ liệu **HttpApplicationState** và **HttpSessionState** nằm ẩn sau.

## 8.12 Tạo một C# Web Application đơn giản

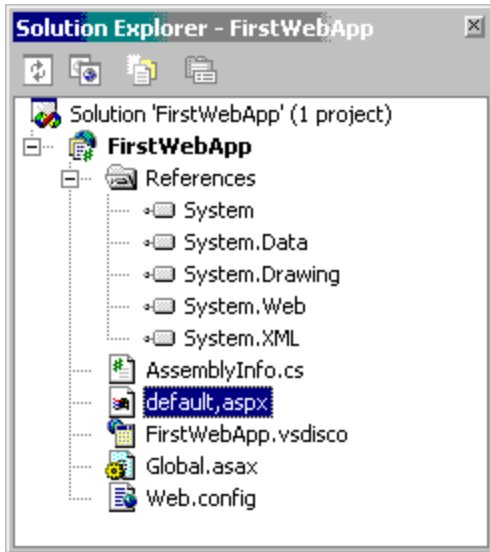
Bây giờ ta tạo một dự án trắc nghiệm nhỏ và thử quan sát cấu trúc tổng quan của ASP.NET Framework. Trước tiên, bạn tạo một dự án C# Web cho mang tên **FirstWebApp**. Bạn ra lệnh **Files | New | Project** rồi chọn **Visual C# Projects** và **Web Application**. Cho đặt tên **FirstWebApp** và cho về đúng thư mục. Xem hình 8-22.

Trước khi click <OK> bạn để ý ô Location trên hình 8-22. Ô này không ánh xạ lên một thư mục đặc biệt trên ổ đĩa cứng mà là URL của máy tính lưu trữ ứng dụng Web này. Các tập tin Visual Studio.NET solution (\*.sln và \*.suo) sẽ được trữ ở thư mục con “MyDocuments\VisualStudio Projects”.

Một khi workspace của dự án mới được tạo xong, bạn sẽ thấy design time template sẽ tự động được mở ra. Xem hình 8-23.



Hình 8-23: Template vào lúc thiết kế



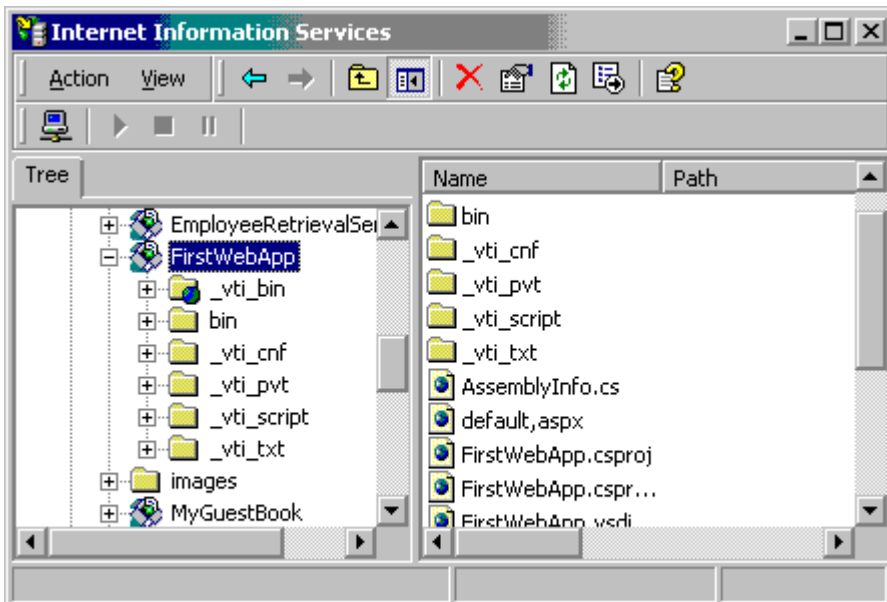
**Hình 8-24: Các tập tin ban đầu của một ứng dụng ASP.NET**

Giống như với ứng dụng Windows Forms, template này tượng trưng cho dáng dấp nhìn thấy được của tập tin \*aspx mà bạn đang xây dựng. Lẽ dĩ nhiên, khác biệt ở đây là bạn đang dùng WebForm controls thiên về HTML thay vì Windows Forms control thiên về Win32. Bạn để ý tên mặc nhiên của trang này là WebForm1. Vì đây là trang yêu cầu bởi thế giới bên ngoài, ta cho đổi lại tên thành default.aspx.

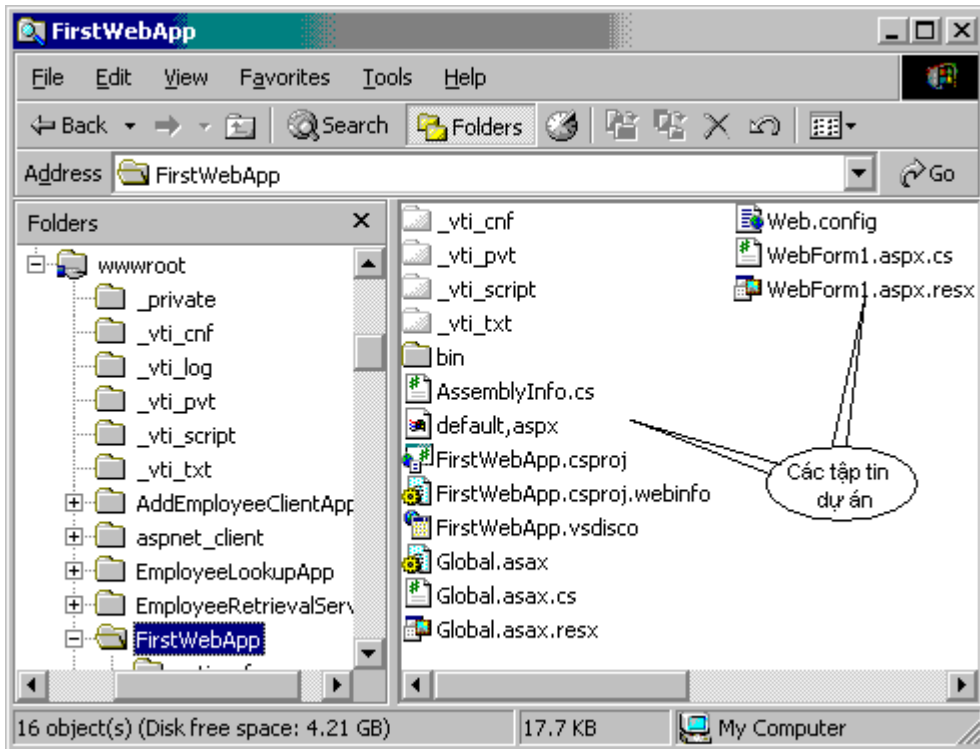
Tiếp theo, bạn nhìn vào cửa sổ Solution Explorer (hình 8-24) bạn thấy một số tập tin mới và những qui chiếu ngoại lai về các assembly.

Nếu bạn mở IIS, bạn sẽ thấy thư mục ảo mới tự động được tạo ra, FirstWebApp. Xem hình 8-25.

Như bạn có thể thấy, mỗi tập tin trong workspace đều được bao gồm trong thư mục ảo này. Thư mục vật lý được ánh xạ đối với thư mục ảo này có thể tìm thấy ở thư mục con <drive>:\Inetpub\wwwroot. Xem hình 8-26.



**Hình 8-25: Thư mục ảo mới (tự động được tạo ra)**



Hình 8-26: Tập tin vật lý chứa các tập tin dự án

## 8.12.1 Thử khảo sát tập tin \*.aspx ban đầu

Nếu bạn thử xem HTML nằm sau tập tin \*.aspx, bạn thấy là bạn đã có một tập hợp tối thiểu những tag để thiết lập một biểu mẫu HTML cơ bản. Điểm lý thú đầu tiên là attribute **runat** xuất hiện ở tag <form> mở đầu. Tag này là tâm điểm và duy nhất của ASP.NET được dùng đánh dấu một item như là ứng viên sẽ được xử lý bởi ASP.NET runtime để kết sinh ra HTML trả về cho browser, như sau:

```
<%@ Page language="c#" Codebehind="default.aspx.cs"
AutoEventWireup="false" Inherits="FirstWebApp.WebForm1" %>

<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0 Transitional//EN" >
<html>
  <head>
    <title>WebForm1</title>
    <meta name="GENERATOR" Content="Microsoft Visual Studio 7.0">
    <meta name="CODE_LANGUAGE" Content="C#">
    <meta name="vs_defaultClientScript" content="JavaScript">
    <meta name="vs_targetSchema" content=
      "http://schemas.microsoft.com/intellisense/ie5">
```

```
</head>
<body MS_POSITIONING="GridLayout">
    <form id="Form1" method="post" runat="server">
    </form>
</body>
</html>
```

Đoạn mã ban đầu thiết lập một số nét liên quan đến trang Web hiện hành. Trước tiên, bạn có thể thấy ngôn ngữ được dùng ở hậu trường để xây dựng trang Web của bạn (ở đây là C#). Attribute **Codebehind** đặt tên cho tập tin C# tượng trưng cho việc xử lý ở hậu trường. Còn attribute **Inherits** dùng khai báo tên của lớp tượng trưng cho lớp được định nghĩa trong tập tin được khai báo bởi Codebehind.

## 8.12.2 Thử khảo sát tập tin Web .Config

Tập tin **Web.config** chứa dữ liệu XML dùng kiểm soát những khía cạnh khác nhau liên quan đến cấu hình của ứng dụng Web. Tập tin này thường nằm ở gốc của thư mục ảo được gắn liền với ứng dụng Web và áp dụng cho mỗi thư mục con. Theo mặc nhiên, tập tin này chứa thông tin liên quan đến biên dịch, sai lầm, an ninh, gỡ rối, phiên giao dịch và globalization (toàn cầu hóa).

## 8.12.3 Thử khảo sát tập tin Global.asax

Giống như với ASP cổ điển, các ứng dụng ASP.NET định nghĩa một tập tin “toàn cầu hóa” (global.asax) cho phép bạn tương tác với các tình huống cấp ứng dụng (và cấp session) cũng như chia sẻ sử dụng các dữ liệu trạng thái thông dụng. Nếu bạn right-click lên tập tin global.asax trên Solution Explorer, rồi chọn click View Code, thì bạn thấy thông tin này được tượng trưng bởi một lớp mang tên Global được dẫn xuất từ lớp cơ bản trừu tượng **HttpApplication**, như theo bảng liệt in sau đây:

```
public class Global: System.Web.HttpApplication
{
    public Global() {InitializeComponent();}

    protected void Application_Start(Object sender, EventArgs e) {}

    protected void Session_Start(Object sender, EventArgs e) {}

    protected void Application_BeginRequest(Object sender,
        EventArgs e) {}

    protected void Application_EndRequest(Object sender,EventArgs e){}

    protected void Application_AuthenticateRequest(
        Object sender, EventArgs e) {}
    protected void Application_Error(Object sender, EventArgs e){}
```

```
protected void Session_End(Object sender, EventArgs e) {}

protected void Application_End(Object sender, EventArgs e) {}
}
```

Trong chừng mực nào đó, lớp **Global** giữ vai trò trung gian giữa client nằm ngoài và Web Form. Nếu bạn đã làm quen ASP cổ điển thì một vài tình huống ở đây quá quen thuộc đối với bạn. Nhìn chung, những tình huống ở đây cho phép bạn đáp ứng việc khởi gán (và kết thúc) của ứng dụng Web và những châu giao dịch riêng biệt.

## 8.12.4 Thêm vài đoạn mã đơn giản C#

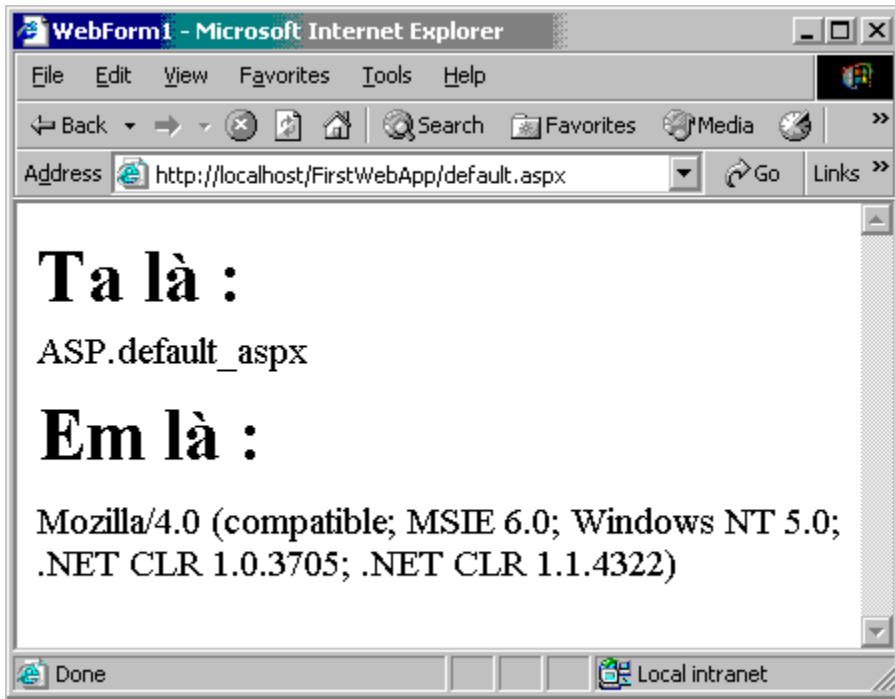
Tới đây, ASP.NET engine trả về cho bạn một trang Web trống rỗng “còn trình bạch”. Để thay đổi tình trạng, ta thử thay đổi phần body của tập tin \*.aspx trả về một vài thông tin dạng văn bản cho biết những khía cạnh khác nhau liên quan đến HTTP request (thuộc tính **System.Web.UI.Page.Response** sẽ được khảo sát chi tiết về sau trong chương này):

```
<body MS_POSITIONING="GridLayout">
  <h1>
    <b> Ta là:</b>
  </h1>
  <%=this.ToString() %>
  <h1>
    <b> Em là:</b>
  </h1>
  <%=Request.ServerVariables["HTTP_USER_AGENT"] %>
  <form id="Form1" method="post" runat="server">
    </form>
</body>
```

Khi khổ xong đoạn mã trên, bạn cho biên dịch và chạy thử. Hình 8-27 cho thấy kết quả trang Web. Trang này sưu liệu ai phát đi yêu cầu cũng như đơn vị nhận yêu cầu (tên trang Web).

Xem ra chức năng của ASP.NET cũng tương tự như với ASP cổ điển. Khác biệt duy nhất là đối tượng **Request** giờ đây là một thuộc tính của lớp cơ bản **Page**. Ngoài ra, như bạn có thể thấy, bạn không viết một đoạn mã kịch bản trong những tag <%...%. mà là đoạn mã C# đăng hoàng chẳng hạn:

```
</h1><%=this.ToString() %><h1>
```



Hình 8-27: Sưu liệu ai là ai

## 8.13 Thử xem kiến trúc của một ứng dụng ASP.NET Web

Bây giờ bạn có thể đào sâu xem kiến trúc của một ứng dụng ASP.NET Web ra sao. Điểm đáng chú ý là thuộc tính bí hiểm **Codebehind** ở đầu khối đoạn mã kịch bản:

```
<%@ Page language="c#" Codebehind="default.aspx.cs"  
AutoEventWireup="false" Inherits="FirstWebApp.WebForm1" %>
```

Khác biệt chủ yếu giữa ASP cổ điển và ASP.NET là trang *\*.aspx*, được yêu cầu bởi một client nằm ngoài, được tượng trưng bởi một lớp C# duy nhất, nhận diện bởi thuộc tính **Codebehind**. Khi một client yêu cầu một trang Web *\*.aspx* đặc biệt nào đó, thì một đối tượng của lớp này sẽ được thể hiện (và được thao tác) bởi ASP.NET Runtime. Tuy nhiên, bạn để ý là tập tin C# này không được cho thấy bởi Solution Explorer. Muốn truy cập tập tin **Codebehind**, chỉ cần right-click lên tập tin *\*.aspx* được mở và chọn click View Code. Sau đây đoạn mã ban đầu:

```
using System;
```

```

using System.Collections;
using System.ComponentModel;
using System.Data;
using System.Drawing;
using System.Web;
using System.Web.SessionState;
using System.Web.UI;
using System.Web.UI.WebControls;
using System.Web.UI.HtmlControls;

namespace FirstWebApp
{
    public class WebForm1: System.Web.UI.Page
    {
        private void Page_Load(object sender, System.EventArgs e)
        {
            // Put user code to initialize the page here
        }

        override protected void OnInit(EventArgs e)
        {
            InitializeComponent();
            base.OnInit(e);
        }

        private void InitializeComponent()
        {
            this.Load += new System.EventHandler(this.Page_Load);
        }
    }
}

```

Đoạn mã mặc nhiên không có chi là rắc rối. Hàm constructor của lớp được dẫn xuất từ **Page** thiết lập một hàm thụ lý tình huống **Init**. Phần thi công hàm thụ lý tình huống này sẽ triệu gọi hàm hành sự **InitializeComponent()**, và hàm này thiết lập một hàm thụ lý tình huống **Load**.

### 8.13.1 Kiểu dữ liệu **System.Web.UI.Page**

Muốn hiểu mục đích của lớp tự kết sinh, ta thử bắt đầu quan sát lớp cơ bản **System.Web.UI.Page**. Lớp **Page** định nghĩa các thuộc tính, hàm hành sự, và tình huống thông dụng đối với tất cả các trang Web được xử lý trên server bởi ASP.NET Runtime. Bảng 8-4 mô tả một vài thuộc tính cốt lõi.

**Bảng 8-4: Các thuộc tính cốt lõi của kiểu dữ liệu *Page***

Các thuộc tính	Mô tả
<b>Application</b>	Đi lấy đối tượng <b>HttpApplication</b> cung cấp bởi Runtime.
<b>Cache</b>	Cho biết đối tượng <b>Cache</b> mà ta cho trữ dữ liệu do ứng dụng trang Web đem lại.

<b>IsPostBack</b>	Đi lấy một trị gì đó cho biết liệu xem trang Web được nạp vào đáp ứng một client postback, hay là được nạp vào và được truy cập lần đầu tiên.
<b>Request</b>	Đi lấy đối tượng <b>HttpRequest</b> cho phép truy cập dữ liệu từ các yêu cầu HTTP mới đến.
<b>Response</b>	Đi lấy đối tượng <b>HttpResponse</b> cho phép bạn gửi đi dữ liệu HTTP trả về cho một client browser.
<b>Server</b>	Đi lấy đối tượng <b>HttpServerUtility</b> được cung cấp bởi HTTP Runtime.
<b>Session</b>	Đi lấy đối tượng <b>System.Web.SessionState.HttpSessionState</b> đem lại thông tin liên quan đến phiên giao dịch của yêu cầu hiện hành.

Như bạn có thể thấy, kiểu dữ liệu **Page** định nghĩa các thuộc tính có dính dáng đến mô hình đối tượng bẩm sinh của ASP cổ điển. Ngoài việc định nghĩa một số hàm hành sự được kế thừa (mà bạn không cần tương tác trực tiếp), lớp **Page** còn đem lại những tình huống critical được mô tả trên bảng 8-5.

**Bảng 8-5: Các tình huống của kiểu dữ liệu Page**

<b>Các tình huống</b>	<b>Mô tả</b>
<b>Init</b>	Tình huống này được phát pháo khi trang Web được khởi gán và đây là trang đầu tiên trong chu kỳ sống của trang Web.
<b>Load</b>	Một khi được khởi gán, tình huống <b>Load</b> sẽ được phát pháo. Ở đây bạn có thể cấu hình hóa bất cứ Web control nào với cảm giác nhìn thấy ban đầu.
<b>Unload</b>	Tình huống này xảy ra khi ô control được giải phóng khỏi ký ức. Các ô control phải được dọn dẹp trước khi cho “đi đong”.

Hàm thụ lý tình huống (event handler) đối với tình huống **Load** là nơi lý tưởng để kết nối với dữ liệu nguồn (để điền dữ liệu vào một WebForm DataGrid) và cho thực hiện bất cứ bước chuẩn bị nào. Còn hàm **Unload** là nơi lý tưởng để dọn dẹp giải phóng các nguồn lực được cấp phát nay hết xài.

## 8.13.2 \*.aspx/Codebehind Connection

Ngoài đoạn mã làm sẵn (boilerplate code), lớp C# được tượng trưng bởi tag **Codebehind** có thể được nói rộng bởi bất cứ số lượng nào các thuộc tính và hàm hành sự có thể được triệu gọi (gián tiếp) bởi khối mã lệnh `<%...%>` trong tập tin \*.aspx của bạn. Như bạn có thể nhớ lại, ASP cổ điển đòi hỏi bạn định nghĩa chức năng custom trực tiếp trong tập tin \*.asp. Do đó, các trang Web của bạn đầy đầy những tag HTML và các đoạn mã VBScript (hoặc JavaScript). Vì vậy các tập tin \*.asp rất khó đọc cũng như khó bảo trì và tái sử dụng.

Bây giờ trên các tập tin \*.aspx một khái niệm hơi kỳ kỳ bắt đầu hình thành khi bạn viết đoạn mã, bạn có thể qui chiếu các hàm hành sự và thuộc tính “cây nhà lá vườn” được định nghĩa trong các tập tin \*.asp. Ta thử lấy một thí dụ đơn giản.



Giả sử bạn muốn xây dựng một hàm hành sự đơn giản lo lấy thời gian và ngày tháng hiện hành. Bạn có thể làm trực tiếp trong lớp được dẫn xuất từ **Page** như sau:

```
public class WebForm1: System.Web.UI.Page
{
    // đoạn mã được kết sinh ở đây...

    public string GetDateTime()
    {
        return DateTime.Now.ToString();
    }
}
```

Muốn qui chiếu hàm hành sự này trong đoạn mã \*.aspx, đơn giản bạn viết như sau:

```
<body>
    <!-- Lấy time từ lớp C# -->
    <% Response.Write(GetDateTime()); %>
    . . .
    <form method="post" runat="server" ID=Form1> </form>
</body>
```

Bạn cũng có thể qui chiếu trực tiếp các thành viên **Page** được kế thừa trong lớp C# của bạn. Do đó, bạn cũng có thể viết như sau:

```
public class WebForm1: System.Web.UI.Page
{
    // đoạn mã được kết sinh ở đây...

    public void GetDateTime()
    {
        Response.Write("Giờ đây là " + DateTime.Now.ToString());
    }
}
```

Và sau đó đơn giản bạn triệu gọi:

```
<!-- Đi lấy thời gian -->
<% GetDateTime(); %>
```

### 8.13.3 Làm việc với Page.Request Property

Như bạn đã thấy trong phần đi trước chương này, dòng chảy cơ bản của một châu giao dịch Web bắt đầu khi khách hàng đăng nhập vào web site, điền thông tin liên quan đến người sử dụng, rồi click nút <Submit> được duy trì bởi một HTML form. Trong đa số trường hợp, tag mở màn của lệnh biểu mẫu cho biết một action và một method cho biết tập tin trên Web server gọi đi dữ liệu lên những ô control HTML khác nhau và hàm hành sự lo gọi đi dữ liệu này (GET hoặc POST). Sau đây là một thí dụ:

```
<form name=MainForm action="http://localhost/default.aspx" method=get
ID=Form1>
```

Trên ASP.NET, thuộc tính **Page.Request** cho phép truy cập dữ liệu mà HTTP request gửi đi. Sau hậu trường, thuộc tính này thao tác một thể hiện của lớp **HttpRequest**. Bảng 8-6 liệt kê một vài thành viên cốt lõi của lớp **HttpRequest** này. Nếu bạn đã rành ASP cổ điển thì bạn không lạ gì các thành viên này.

**Bảng 8-6: Các thành viên cốt lõi của lớp *HttpRequest***

Các thành viên	Mô tả
<b>ApplicationPath</b>	Đi lấy virtual path chỉ về server thì hành ứng dụng hiện hành
<b>Browser</b>	Cho biết thông tin liên quan đến khả năng của browser client
<b>ContentType</b>	Cho biết kiểu dữ liệu nội dung MIME của incoming request. Thuộc tính này là read-only.
<b>Cookies</b>	Đi lấy một collection các biến cookie của client.
<b>FilePath</b>	Cho biết lối tìm về ảo của request hiện hành. Thuộc tính này là read-only.
<b>Files</b>	Đi lấy collection các tập tin client-uploaded (multipart MIME format).
<b>Filter</b>	Đi lấy hoặc đặt để một bộ lọc dùng khi đọc input stream hiện hành.
<b>Form</b>	Đi lấy một collection các biến Form.
<b>Headers</b>	Đi lấy một collection các tiêu đề HTTP.
<b>HttpMethod</b>	Cho biết hàm hành sự chuyển dữ liệu HTTP mà client dùng (GET, POST). Thuộc tính này là read-only.
<b>IsSecureConnection</b>	Cho biết liệu xem HTTP Connection có an toàn hay không (nghĩa là HTTPS). Thuộc tính này là read-only.
<b>Params</b>	Đi lấy một collection được phối hợp QueryString + Form + ServerVariable + Cookies.
<b>QueryString</b>	Đi lấy collection các biến QueryString.
<b>RawUrl</b>	Đi lấy raw URL của request hiện hành.
<b>RequestType</b>	Cho biết hàm hành sự chuyển dữ liệu HTTP mà client dùng đến (GET, POST).
<b>ServerVariables</b>	Đi lấy một collection các biến của Web server.
<b>UserHostAddress</b>	Đi lấy địa chỉ IP host của client nằm ở xa.
<b>UserHostName</b>	Đi lấy tên DSN của client nằm ở xa.

Bạn đã thấy các thành viên của lớp **HttpRequest** trước đây trong chương này. Thí dụ, khi bạn muốn biết các đặc tính của HTTP request đi vào, bạn quen nhìn vào đối tượng **Request** như sau:

```
<b>Em là: </b><%=Request.ServerVariables["HTTP_USER_AGENT"] %>
```

Việc bạn đang thực sự làm là truy cập một thuộc tính trên kiểu dữ liệu **HttpRequest** được trả về, như sau:

```
<b>Em là: </b>
<%
    HttpRequest r;
    r = this.Request;
    Response.Write(r.ServerVariables["HTTP_USER_AGENT"]);
%>
```

Bây giờ ta thử khảo sát thuộc tính **Request.Response** và kiểu dữ liệu **HttpResponse** liên đới.

### 8.13.4 Làm việc với thuộc tính Page.Response

Thuộc tính **Page.Response** cho phép truy cập vào một kiểu dữ liệu **HttpResponse** nội tại. Lớp **HttpResponse** này định nghĩa một số thuộc tính cho phép bạn định dạng trả lời HTTP được gửi trả về cho client browser. Bảng 8-7 cho liệt kê một vài thuộc tính cốt lõi.

**Bảng 8-7: Các thuộc tính cốt lõi của lớp *HttpResponse***

Các thuộc tính	Mô tả
<b>Cache</b>	Trả về caching semantics của trang Web (nghĩa là expiration time, privacy, vary clauses)
<b>ContentEncoding</b>	Đi lấy hoặc đặt để bộ ký tự HTTP của kết xuất.
<b>ContentType</b>	Đi lấy hoặc đặt để kiểu dữ liệu HTTP MIME của kết xuất
<b>Cookies</b>	Đi lấy collection <b>HttpCookie</b> được gửi đi bởi request hiện hành.
<b>Filter</b>	Cho biết đối tượng wrapping fileter để thay đổi HTTP entity body trước khi chuyển tin.
<b>IsClientConnected</b>	Đi lấy một trị cho biết liệu xem client có còn được kết nối hay không.
<b>Output</b>	Cho phép custom output về nội dung phần thân của outgoing HTTP.
<b>OutputStream</b>	Cho phép binary output về nội dung phần thân của outgoing HTTP.
<b>StatusCode</b>	Đi lấy hoặc đặt để mã trạng thái HTTP của kết xuất được trả về cho client.
<b>StatusDescription</b>	Đi lấy hoặc đặt để chuỗi trạng thái HTTP của kết xuất được trả về cho client.
<b>SuppressContent</b>	Đi lấy hoặc đặt để một trị cho biết nội dung HTTP sẽ không được gửi cho client.

Bảng 8-8 mô tả các hàm hành sự của lớp **HttpResponse**. Có thể, khía cạnh quan trọng nhất của lớp **HttpResponse** là khả năng viết ra HTTP output stream. Như bạn có thể thấy, bạn có thể trực tiếp triệu gọi hàm **Write()** hoặc inline một output request sử dụng ký hiệu `<%=...%>` (giống như với ASP cổ điển).

**Bảng 8-8: Các hàm hành sự của lớp *HttpResponse***

Các hàm hành sự	Mô tả
<b>AppendHeader()</b>	Thêm một HTTP header vào output stream.
<b>AppendToLog()</b>	Thêm một custom log info vào tập tin IIS log.
<b>Clear()</b>	Xoá trắng tất cả các header và nội dung output khỏi buffer stream.
<b>Close()</b>	Cho đóng lại socket kết nối về một client.
<b>End()</b>	Cho chuyển đi về cho client tất cả các output hiện trong vùng đệm rồi cho đóng lại socket kết nối.
<b>Flush()</b>	Cho chuyển đi về cho client tất cả các output hiện trong vùng đệm
<b>Redirect()</b>	Chuyển hướng một client về một URL mới.
<b>Write()</b>	Viết các trị ra một HTTP output content stream.
<b>WriteFile()</b>	Overloaded. Viết trực tiếp một tập tin lên một HTTP content output stream.

như vậy, bạn có thể thao tác đối tượng này từ tập tin \*.aspx như sau:

```
<b>Em là: </b>
<%
    HttpRequest r;
    r = this.Request;

    HttpResponse rs;
    rs = this.Response;

    rs.Write(r.ServerVariables["HTTP_USER_AGENT"]);
%>
```

Đoạn mã trên tương đương dòng lệnh sau đây:

```
<% Request.ServerVariables["HTTP_USER_AGENT"] %>
```

### 8.13.5 Làm việc với thuộc tính **Page.Application**

Thuộc tính **Page.Application** cho phép truy cập lớp **HttpApplicationState** nằm ẩn sau. Như đã đề cập đến trước đây, lớp **HttpApplicationState** cho phép các nhà triển khai chia sẻ sử dụng các thông tin mang tính toàn cục xuyên qua các châu giao dịch trong một ứng dụng ASP.NET. Bảng 8-9 mô tả các thuộc tính cốt lõi của lớp **HttpApplicationState** này.

Khi bạn cần tạo một thành viên dữ liệu có thể được chia sẻ sử dụng bởi tất cả các châu giao dịch hiện dịch, bạn chỉ cần thiết lập một cặp name/value (chẳng hạn firstUser = duong) rồi cho chèn nó vào **KeysCollection** được duy trì trong nội bộ. Muốn thế, bạn sử dụng một class indexer, như sau:

**Bảng 8-9: Các thuộc tính cốt lõi của lớp *HttpApplication State***

Các thuộc tính	Mô tả
<b>AllKeys</b>	Cho phép người sử dụng tìm thấy lại tên đối tượng trạng thái ứng dụng trong một collection.
<b>Count</b>	Đi lấy số lượng các đối tượng item trong một collection trạng thái ứng dụng (application state)
<b>Keys</b>	Trả về một thể hiện <b>NameObjectCollectionBase.KeysCollection</b> chứa tất cả các mục khoá trên thể hiện <b>NameObjectCollection Base</b> .
<b>StaticObjects</b>	Cho trưng ra tất cả các đối tượng được khai báo thông qua một tag <x runat=server></x> trong tập tin ứng dụng ASP.NET.

```

public class WebForm1: System.Web.UI.Page
{
    protected void Page_Load(object sender, EventArgs e)
    {
        if(!IsPostBack)
        {
            // Tạo một thành viên dữ liệu cấp ứng dụng
            Application["AppString"] = "Initial App Value";
        }
    }
}

```

Về sau, khi bạn cần qui chiếu trị này, bạn chỉ cần trích nó ra bằng cách dùng cùng thuộc tính như sau:

```
string appVar = "App: " + Application["AppString"];
```

## 8.13.6 Làm việc với thuộc tính Page.Session

Như đã phát biểu trước đây, một *phiên giao dịch* là một lúc tương tác của người sử dụng với một ứng dụng Web. Muốn duy trì những thông tin bổ ích đối với một người sử dụng đặc biệt nào đó, bạn sử dụng thuộc tính **Session**. Bạn có thể tha hồ quan sát thuộc tính này trên MSDN.

## 8.14 Gỡ rối và Lần theo dấu vết trên các ứng dụng ASP.NET Web

Khi bạn tạo những dự án ASP.NET Web bạn có thể dùng cùng kỹ thuật gỡ rối (debugging) như với tất cả các dự án .NET khác. Như vậy bạn có thể đặt dễ những chốt ngừng (breakpoint) (kể cả bất cứ tập tin lớp C# nào), bắt đầu một chuỗi gỡ rối (ấn F5) và đi qua từng bước một trong đoạn mã (hình 8-28):



Hình 8-28: Thiết lập các breakpoint

Cũng thế, bạn có thể có những hỗ trợ lần theo dấu vết (tracing) đối với các tập tin \*.aspx bằng cách khai báo thuộc tính **trace** trong khối kịch bản mở màn như sau:

```

<%@ Page language="c#" Codebehind="default.aspx.cs"
AutoEventWireup="false" Inherits="FirstWebApp.WebForm1" trace=true %>
  
```

Khi bạn làm như thế, thì HTML được trả về sẽ chứa những thông tin theo dõi dấu vết liên quan đến trả lời HTTP đi trước. Xem hình 8-29.

Muốn cho chèn các thông điệp theo dõi dấu vết của riêng bạn, bạn có thể dùng Trace type. Bất cứ lúc nào bạn muốn log một custom message (từ một khối kịch bản hoặc đoạn mã nguồn C#) bạn chỉ cần triệu gọi hàm **Write()**, (hình 8-30) như sau:

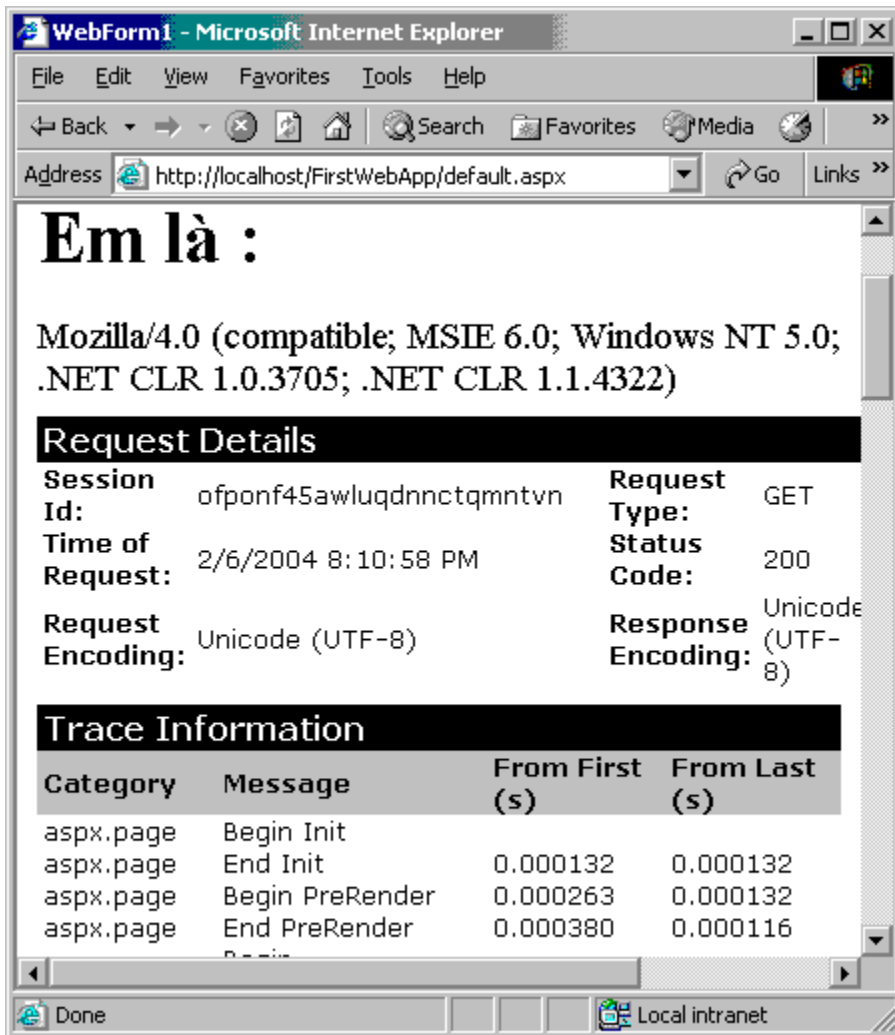
```

<b>Em là: </b>
<%
    Trace.Write("App Category", "Liên quan xác định agent...");
    HttpRequest r;
    r = this.Request;

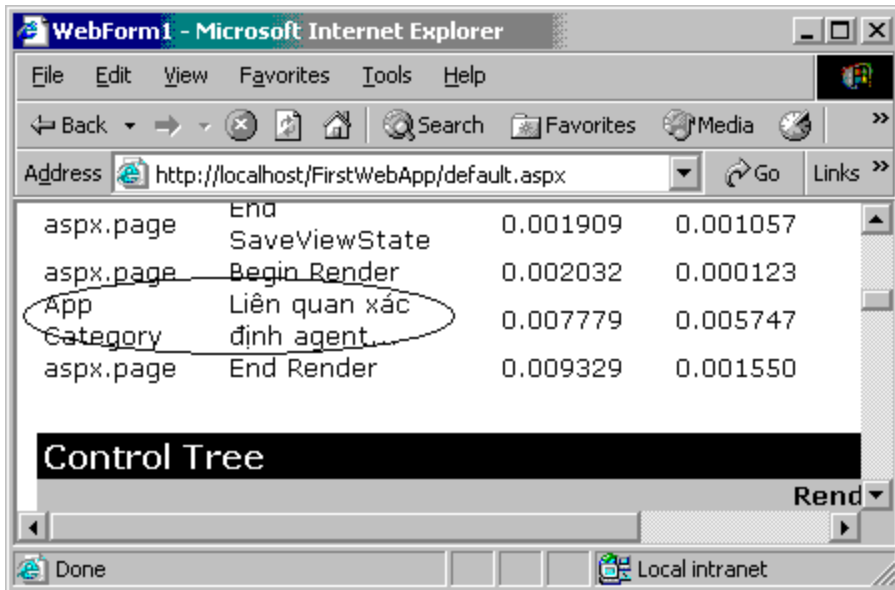
    HttpResponse rs;
    rs = this.Response;
  
```

```
rs.Write(r.ServerVariables["HTTP_USER_AGENT"]);
%>
```

Bây giờ, ta thử làm một tua du lịch đối với các ô control GUI Web (thường được gọi là GUI widget).



Hình 8-29: Cho phép có thông tin Trace



Hình 8-30: Cho log các thông điệp custom

## 8.15 Tìm hiểu các ô WebForm Controls

Một lợi điểm chính của ASP.NET là khả năng ráp nối giao diện người sử dụng (GUI) trên các trang Web sử dụng các ô control được định nghĩa trong namespace **System.Web.UI.WebControls**, nghĩa là bạn có thể thêm những ô control phía server vào một Web Form. Các ô control này (thường được gọi là server control, Web control hoặc Web form control) rất hữu ích vì chúng tự động được kết sinh với những tag HTML cần thiết đối với browser.

Thí dụ, trên ASP cổ điển, nếu bạn tạo một trang Web cần hiển thị một loạt ô control text box, cơ bản bạn phải khổ trực tiếp vào tag HTML lên trang ASP. Tuy nhiên, trên ASP.NET bạn chỉ cần thiết kế biểu mẫu Web sử dụng design time template và các ô control WebForm bẩm sinh. Bạn có hai cách để đưa các ô control này vào Web Form: bằng tay, bằng cách viết HTML vào trang HTML, hoặc lôi các ô control từ Toolbox thả lên trang Design. Thí dụ, giả sử bạn muốn sử dụng những nút radio button cho phép người sử dụng chọn một trong 3 nút, bạn có thể viết HTML vào phần tử <form> trên cửa sổ HTML như sau:

```
<asp:RadioButton GroupName="Shipper" id="Airborne"
    text="Airborne Express" Checked="True" runat="server">
</asp:RadioButton>
<asp:RadioButton GroupName="Shipper" id="UPS"
    text="United Parcel Service" runat="server">
```



```

</asp:RadioButton>
<asp:RadioButton GroupName="Shipper" id="Federal"
    text="Federal Express" runat="server">
</asp:RadioButton>

```

Còn nếu bạn muốn lôi thả vào biểu mẫu một ô text box sử dụng Toolbox, thì sẽ kết sinh như sau:

```

<form id="Form1" method="post" runat="server">
    <asp:TextBox id="TextBox1" style="Z-INDEX: 101; LEFT: 20px;
        POSITION: absolute; TOP: 195px" runat="server"
        Width="97px" Height="43px">
    </asp:TextBox>
</form>

```

Bạn thấy hai đoạn mã kể trên, đoạn viết bằng tay và đoạn được kết sinh, khác nhau ở các thuộc tính cho biết vị trí và kích thước của các ô control. Bạn có thể suy đoán ra là đoạn mã viết bằng tay sẽ được thêm ở hậu trường bởi những thuộc tính vị trí và kích thước theo những trị mặc nhiên.

Khi ASP.NET Runtime gặp phải ô control text box với attribute này, HTML bình thường sẽ được đưa tự động vào response stream như sau:

```

<input name="TextBox1" type="text" id="TextBox1" style="Z-INDEX: 101;
LEFT: 27px; POSITION: absolute; TOP: 30px" />

```

Bạn thấy là tag **asp:** khai báo các ô control ASP.NET phía server sẽ được thay thế bởi HTML bình thường khi server xử lý trang. Trong trường hợp này, xem ra WebForm control đòi hỏi nhiều markup hơn là định nghĩa raw HTML widget. Tuy nhiên, không phải tất cả các ô control đều là tầm thường như là ô text box hay radio button. Thí dụ, vài Web control bao gồm một tấm lịch khá hấp dẫn, ad rotator, HTML table, data grid, v.v.. Trong trường hợp này, WebForm control có thể tiết kiệm cho bạn vài tá lệnh HTML thô.

Một lợi điểm khác là mỗi ASP.NET control đều có một lớp tương ứng trong namespace **System.Web.UI.WebControls** và như vậy có thể xử lý bằng lập trình từ các tập tin \*.aspx của bạn cũng như với lớp được dẫn xuất từ **Page** được gắn liền (nghĩa là lớp C# được đánh dấu bởi thuộc tính Codebehind). Ngoài ra, Web control cũng có một số tính huống có thể được xử lý trên server.

Một lợi điểm cốt lõi khác khi sử dụng WebForm control (thay vì raw HTML control) là trong thực tế ASP.NET cung cấp một tập hợp những ô control để kiểm tra hợp lệ dữ liệu người sử dụng cung cấp. Do đó, bạn khỏi phải kết sinh những trình kiểm tra hợp lệ từ phía client sử dụng JavaScript (mặc dù bạn tùy nghi làm thế).

## 8.16 Làm việc với các ô WebForm Controls

Khi bạn xây dựng những dự án Web, bạn sẽ thấy một hộp đồ nghề (toolbox) dành cho Web Forms. Bạn chỉ cần gọi Toolbox, rồi click tab Web Forms, thì bạn thấy hiện lên một số ô control tương tự như với Windows Forms.

Mỗi ô control Web Forms có thể được cấu hình hóa sử dụng cửa sổ **Properties**. Chúng tôi thiết nghĩ bạn đã biết cách sử dụng cửa sổ này.

Khi bạn cấu hình hóa một WebControl nào đó sử dụng đến cửa sổ Properties, những thay đổi này sẽ được phản ánh trực tiếp lên tập tin \*.aspx. Thí dụ, nếu bạn chọn ô control **TextBox** (mang tên txtEmail) và thay đổi **BorderStyle**, **BorderWidth**, **BackColor**, **BorderColor** và **ToolTip**, thì tag mở đầu <asp:Textbox> có một số cặp name/value mới phản ánh những lựa chọn của bạn:

```
<form id="Form1" method="post" runat="server">
  <asp:TextBox id="txtEmail" style="Z-INDEX: 101; LEFT: 20px;
    POSITION: absolute; TOP: 195px" runat="server"
    Width="97px" Height="43px" BorderStyle="Ridge"
    BorderWidth="5px" BackColor="PaleGreen"
    BorderColor="DarkOliveGreen" ToolTip="Khô e-mail vào đây ...">
  </asp:TextBox>
</form>
```

Một lần nữa, viết ra theo HTML cũ xưa như sau:

```
<input name="txtEmail" type="text" value="fdfdf" id="txtEmail"
  title="Khô e-mail vào đây ..."
  style="background-color:PaleGreen;border-color:DarkOliveGreen;
    border-width:5px;border-style:Ridge;" />
```

Bây giờ ta thử xem đúng ra các ô control WebForms này được tượng trưng trong tập tin \*.aspx thế nào. Một WebControl nào đó sẽ được định nghĩa sử dụng cú pháp giống XML với tag mở đầu bao giờ cũng là <asp: *controlType* runat="server">. Còn tag đóng lại sẽ đơn giản là </asp:*controlType*>. Như vậy, bạn sẽ thấy mỗi ô WebControl sẽ được tượng trưng trong tập tin \*.aspx sử dụng cú pháp như sau:

```
<asp:TextBox id=TextBox1 style="Z-INDEX: 101; LEFT: 27px;
  POSITION: absolute; TOP: 30px" runat="server">
</asp:TextBox>

<asp:Button id=Button1 style="Z-INDEX: 102; LEFT: 26px;
  POSITION: absolute; TOP: 66px" runat="server"
  DESIGNTIMEDRAGDROP="21" Text="Button">
</asp:Button>
```

Attribute **runat="server"** đánh dấu item này như là một ô control phía server và báo cho ASP.NET Runtime biết là item này cần được xử lý trước khi trả về cho response stream cho browser, cho kết sinh HTML cần thiết. Bây giờ, bạn cho mở lớp CodeBehind. Bạn sẽ để ý là bạn có một số biến thành viên mới tượng trưng cho mỗi ô control. Như bạn có thể thấy, tên các biến này giống tên phần tử ID được định nghĩa trên tập tin \*.aspx:

```
public class WebForm1: System.Web.UI.Page
{
    protected System.Web.UI.WebControls.TextBox txtLastName;
    protected System.Web.UI.WebControls.TextBox txtFirstName;
    protected System.Web.UI.WebControls.CheckBox ckBoxNewsLetter;
    protected System.Web.UI.WebControls.Button btnSubmit;
    protected System.Web.UI.WebControls.TextBox txtEmail;
    ...
}
```

Theo cách này, bạn có thể lập trình đối với các ô control Web sử dụng đoạn mã C# trong tập tin \*.aspx hoặc trong các thường trình custom trong các lớp được dẫn xuất từ Page.

Bạn có thể thêm những ô control vào trang Web theo một trong hai mode. Mode mặc nhiên là **LinearLayout** (bố trí theo kiểu tuyến tính). Khi bạn thêm vào ô control theo **LinearLayout** thì các ô control này sẽ được hiển thị bởi browser ô này sau ô kia. Bạn sẽ chịu trách nhiệm thêm HTML để hỗ trợ việc canh vị trí của các ô control này.

Mode thay thế là **GridLayout** (bố trí theo kiểu khung lưới). Với **GridLayout**, các ô control được thêm vào một khung lưới, cho phép canh vị trí chính xác hơn. Khi bạn cho **GridLayout** về On, wizard sẽ thêm một table vào trang web cho phép canh chính xác vị trí. Muốn thay đổi từ **GridLayout** qua **LinearLayout** hoặc ngược lại, bạn chỉ cần thay đổi thuộc tính **pageLayout** của tài liệu trên cửa sổ **Properties**.

Web Form cung cấp cho bạn hai loại server-side control. Loại thứ nhất là server-side HTML control, còn được gọi là Web Controls. Đây là những ô control HTML chuẩn mà bạn cho gán tag với attribute **runat=Server**. Loại thứ hai thay thế Web controls là ASP.NET Server Controls, còn được gọi là ASP Controls. ASP Controls được thiết kế để thay thế các ô control HTML chuẩn, cung cấp một mô hình đối tượng nhất quán hơn và có những attribute được đặt tên nhất quán hơn. Thí dụ, với HTML controls, có vô số cách để thụ lý nhập liệu:

```
<input type = "radio">
<input type = "checkbox">
<input type = "button">
<input type = "text">
<textarea>
```

Các ô control trên hành xử một cách khác nhau cũng như lấy những attributes khác

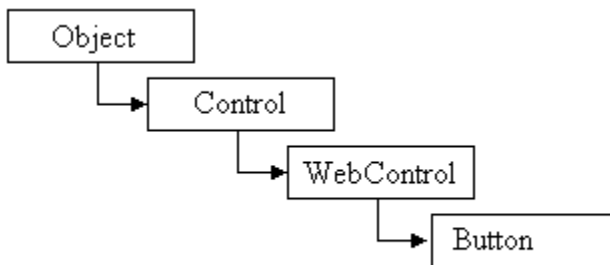
nhau. Còn ASP Controls thì cố normalize tập hợp các ô control, sử dụng nhất quán các attribute xuyên suốt mô hình đối tượng ASP control. Các ASP Control tương ứng với các ô control HTML server side kể trên như sau:

```
<asp:RadioButton>
<asp:Checkbox>
<asp:Button>
<asp:TextBox rows="1">
<asp:TextBox rows="5">
```

Từ đây trở đi chúng tôi chỉ tập trung đề cập đến ASP Controls còn được gọi là WebForm control.

## 8.16.1 Dẫn xuất các ô WebForm Controls

Tất cả các ô control ASP.NET phía server được dẫn xuất từ một lớp cơ bản thông dụng mang tên **System.Web.UI.WebControls**. WebControl lại đến phiên được dẫn xuất từ **System.Web.UI.WebControls.Control**, rồi lại đến phiên được dẫn xuất từ **System.Object**. Thí dụ, dẫn xuất của lớp WebForm Button sẽ được hiểu theo hình 8-31:



**Hình 8-31: Các lớp cơ bản của một WebControl**

giúp bạn hiểu sâu các chức năng được kế thừa, ta thử xem tập hợp con các thuộc tính lớp **Control** được mô tả ở bảng 8-10:

**Bảng 8-10: Các thuộc tính của lớp cơ bản Control**

Các thuộc tính	Mô tả
<b>ID</b>	Đi lấy hoặc đặt để mã nhận diện đối với ô control. Cho đặt để thuộc tính trên một ô control cho phép truy cập bằng lập trình các thuộc tính của ô control cũng như có cơ may đáp ứng những tình huống được gởi đi bởi ô control.
<b>MaintainState</b>	Đi lấy hoặc đặt để một trị cho biết liệu xem ô control phải duy trì view state của nó hay không, và view state của bất cứ ô control con-cái nào mà nó chứa chấp khi request của trang hiện hành kết thúc.
<b>Page</b>	Đi lấy đối tượng <b>Page</b> chứa chấp ô control hiện hành.
<b>Visible</b>	Đi lấy hoặc đặt để một trị cho biết liệu xem một ô control phải hiển thị lên trang Web hay không.

Như bạn có thể nói, lớp **Control** đem lại một số hành xử không liên quan đến GUI. Lớp **WebControl** cũng định nghĩa một vài thuộc tính bổ sung cho phép bạn cấu hình hóa đáng dấp của ô control phía server, như theo bảng 8-11:

**Bảng 8-11: Các thuộc tính của lớp cơ bản WebControl**

Các thuộc tính	Mô tả
<b>BackColor</b>	Đi lấy hoặc đặt để màu nền của ô control Web.
<b>BorderColor</b>	Đi lấy hoặc đặt để màu đường viền của ô control Web.
<b>BorderStyle</b>	Đi lấy hoặc đặt để phong cách của ô control Web.
<b>BorderWidth</b>	Đi lấy hoặc đặt để bề dày đường viền của ô control Web.
<b>Enabled</b>	Đi lấy hoặc đặt để một trị cho biết liệu xem ô control Web có hiệu lực hay không.
<b>Font</b>	Đi lấy thông tin phong chữ liên quan đến ô control Web.
<b>ForeColor</b>	Đi lấy hoặc đặt để màu văn bản của ô control Web.
<b>Height</b> <b>Width</b>	Đi lấy hoặc đặt để chiều cao và khổ rộng của ô control Web
<b>TabIndex</b>	Đi lấy hoặc đặt để chỉ mục Tab của ô control Web.
<b>ToolTip</b>	Đi lấy hoặc đặt để tool tip đối với ô control Web dùng cho hiển thị khi con nháy nằm trên ô control Web.

## 8.17 Các loại ô WebForm Controls

Bạn có thể chia các ô control WebForm theo chức năng thành 4 nhóm:

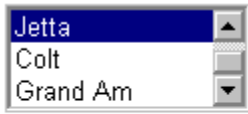
- Các ô control bẩm sinh (intrinsic control)
- Rich controls
- Các ô control thiên về dữ liệu (Data-centric controls)
- Các ô control kiểm tra hợp lệ dữ liệu nhập (input validation control)

Có thể bạn đã từng làm quen với các ô control chạy trên Windows Forms, thì bạn sẽ không ngỡ ngàng gì với các ô control WebForm; chúng sẽ gói ghém việc kết sinh những tag HTML cũng tương tự như ô control Windows Form gói ghém lại các hàm “thô ráp” Win32 API khỏi cái nhìn của bạn.

### 8.17.1 Làm việc với những ô control WebForm “bẩm sinh”

Để bắt đầu, ta thử xét đến vài ô control Web “bẩm sinh”. Những kiểu dữ liệu này điển hình là những .NET component có những đối tác HTML trực tiếp. Thí dụ, muốn cho

hiển thị một danh sách các items cho người sử dụng xem (hình 8-32), bạn phải tạo một WebForm ListBox (và các ListItem có liên hệ) như sau:



**Hình 8-32:: Tạo một ListBox**

```
<asp:ListBox id="ListBox1" style="Z-INDEX: 106;
LEFT: 419px; POSITION: absolute; TOP: 120px"
runat="server" Height="57px" Width="123px">
<asp:ListItem Value="BWM">BWM</asp:ListItem>
<asp:ListItem Value="Jetta">Jetta</asp:ListItem>
<asp:ListItem Value="Colt">Colt</asp:ListItem>
<asp:ListItem Value="Grand Am">Grand
Am</asp:ListItem>
</asp:ListBox></form>
```

Khi các ô control được xử lý bởi ASP.NET Runtime, HTML kết xuất (được hiển thị trên browser) sẽ giống như sau:

```
<select name="ListBox1" id="ListBox1" size="5"
style="height:69px;width:86px;>
<option value="BWM">BWM</option>
<option value="Jetta">Jetta</option>
<option value="Colt">Colt</option>
<option value="Grand Am">Grand Am</option>
</select>
```

Bảng 8-12 mô tả một vài WebForm control bẩm sinh. Làm việc với những ô control này cũng tương tự như với các ô control trên Windows Forms. Công việc của bạn ngày càng nhẹ nhàng hơn khi Visual Studio.NET IDE cung cấp cho bạn cửa sổ **Properties** để cấu hình hóa các ô control được chọn. Do đó, thay vì xem từng ô control bẩm sinh một, ta thử dành thời gian xem một vài cấu hình thông dụng, phổ biến.

**Bảng 8-12: Các ô control WebForms bẩm sinh**

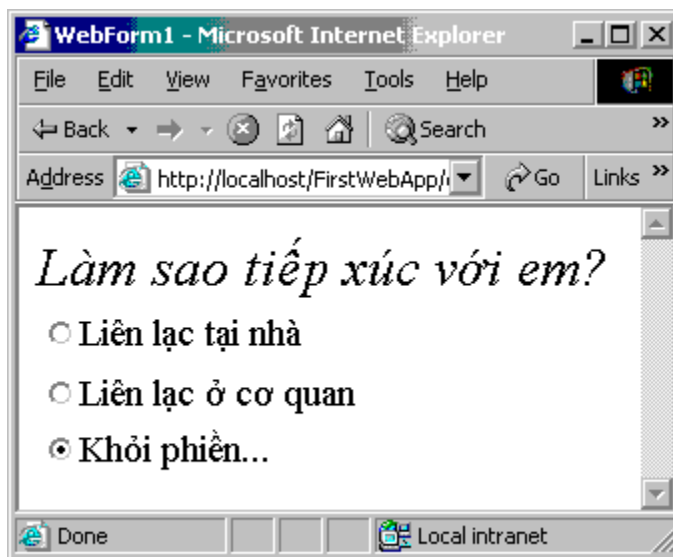
Các ô control	Mô tả
<b>Button</b> <b>Image Button</b>	Các kiểu nút ấn khác nhau
<b>CheckBox</b> <b>CheckBoxList</b>	Một ô duyệt cơ bản (CheckBox) hoặc một ô liệt kê chứa một nhóm ô duyệt (CheckBoxList)
<b>DropDownList</b> <b>ListBox, ListItem</b>	Các kiểu dữ liệu này cho phép tạo những ô liệt kê các mục tin chuẩn.
<b>Image</b> <b>Panel, Label</b>	Các kiểu dữ liệu này tượng trưng cho ô chứa đoạn văn bản static và hình ảnh (cũng như cách gộp chúng với nhau).
<b>RadioButton</b> <b>RadioButtonList</b>	Một nút đài cơ bản (RadioButton) hoặc một ô liệt kê chứa một nhóm nút đài (RadioButtonList)
<b>TextBox</b>	Ô văn bản dành khô vào dữ liệu nhập. Có thể được cấu hình hóa như là ô văn bản một hàng hoặc nhiều hàng.

### 8.17.1.1 Tạo một nhóm ô Radio Buttons

Thường người ta gộp các radio button thành nhóm theo đây chỉ một nút sẽ được chọn mà thôi vào một lúc nào đó. Hình 8-33 cho thấy một nhóm radio button như thế và được viết như sau:

```
<body>
  <form method="post" runat="server">
    <p><font size=5><em>Làm sao tiếp xúc với em?</em></font></p>
    <p><asp:RadioButton id="RadioHome" runat="server"
      text="Liên lạc tại nhà" GroupName="ContactGroup">
    </asp:RadioButton></p>
    <p><asp:RadioButton id="RadioWork" runat="server"
      text="Liên lạc ở cơ quan" GroupName="ContactGroup">
    </asp:RadioButton></p>
    <p><asp:RadioButton id="RadioDontBother" runat="server"
      text="Khỏi phiền..." GroupName="ContactGroup">
    </asp:RadioButton></p>
  </form>
</body>
```

Bạn để ý mỗi Radio Button type đều có một thuộc tính **GroupName**. Khi mỗi item được ánh xạ lên cùng **GroupName** (ở đây là ContactGroup) nên mỗi item sẽ loại trừ lẫn nhau.



Hình 8-33: Tạo một nhóm radio button có liên hệ

### 8.17.1.2 Tạo một ô TextBox nhiều hàng và rào xem được

Một ô control khá phổ biến là multiline text box (hình 8-34).

Như bạn có thể chờ đợi, cấu hình hóa một ô control TextBox trong lúc này chỉ là vấn đề thêm đúng thuộc tính vào tag mở đầu `<asp:TextBox>`. Thí dụ, ta thử xem đoạn mã kịch bản sau đây:

```
<p><font size=3><b>Cho biết việc gì thế:</b></font></p>
<p><asp:TextBox ID=TextBox1 Runat="server" Width="183" Height="96"
    TextMode="MultiLine" BorderStyle="Ridge">
</asp:TextBox></p>
```

Khi bạn cho thuộc tính TextMode về MultiLine, thì ô control TextBox sẽ tự động cho hiển thị một thanh rào đứng (vertical scrollbar) khi nội dung lớn hơn mặt bằng hiển thị.



Hình 8-34: Một ô multiline TextBox

## 8.17.2 Các ô Rich Controls

Rich control cũng là những ô control phát ra HTML lên HTTP response stream. Khác biệt giữa Rich control và các ô control bẩm sinh ở chỗ các ô Rich control không có tương đương trên HTML. Bảng 8-13 cho liệt kê các ô Rich Control:



**Bảng 8-13: Các ô Rich Control WebForm**

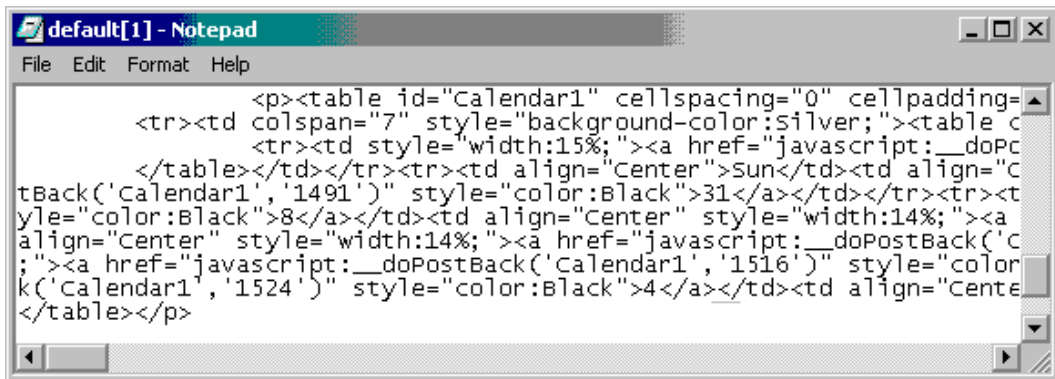
WebForm Rich Control	Mô tả
<b>AdRotator</b>	Ô control này cho phép bạn hiển thị văn bản/hình ảnh một cách hồ đồ sử dụng một tập tin cấu hình XML tương ứng.
<b>Calendar</b>	Ô control này trả về HTML tượng trưng cho một cái lịch GUI.

### 8.18.2.1 Làm việc với Calendar Control

Calendar control là một widget không có tương đương HTML trực tiếp. Tuy nhiên, ô control này được thiết kế trả về một lô tag HTML mô phỏng một thực thể như thế. Thí dụ, giả sử bạn cho đặt một ô control Calendar lên WebForm như sau:

```
<p>
<asp:Calendar id=Calendar1 runat="server"></asp:Calendar>
</p>
```

Muốn tự trắc nghiệm, bạn cho đặt ô control Calendar vào khung thiết kế, cho cất trữ tập tin \*.aspx, rồi cho chạy trên browser. Khi hình cái lịch hiện lên, bạn ra lệnh **View | Source**, thì nội dung HTML của ô control Calendar hiện lên trên Notepad. Xem hình 8-35.

**Hình 8-35: Calendar Web Control phát ra một HTML phức tạp**

Hình 8-36 cho thấy ô control Calendar:

Giống như đối tác Windows Forms của nó, ô control Calendar phía server có thể cắt xén (customizable) tùy thích. Một thành viên đáng lưu ý là thuộc tính SelectionMode. Theo mặc nhiên, ô control Calendar chỉ cho phép người sử dụng chọn một ngày (nghĩa là SelectionMode=Day). Bạn có thể thay đổi cách hành xử này bằng cách gán thuộc tính này về bất cứ thay thế sau đây:



Hình 8-36: Calendar UI phía client

- **None:** Không có chọn lựa (nghĩa là ô control Calendar được hiện lên với mục đích cho thấy mà thôi).
- **DayWeek:** người sử dụng có thể chọn một ngày hoặc trọn tuần lễ.
- **DayWeekMonth:** người sử dụng có thể chọn một ngày, trọn một tuần lễ, hoặc trọn một tháng.

Thí dụ, nếu bạn chọn DayWeekMonth, thì ô control Calendar sẽ hiện lên như theo hình 8-36 với một cột tận cùng bên trái (cho phép người sử dụng chọn một tuần lễ) cũng như một selector trên góc cao tay trái (cho phép người sử dụng chọn hết một tháng). Sau đây là trọn vẹn một cấu hình hóa khá bắt mắt sử dụng cửa sổ Properties trên Visual Studio.NET IDE:

```
<form method="post" runat="server">
<p><asp:calendar id="Calendar1" Runat="server"
    SelectionMode="DayWeekMonth" BackColor="White"
    DayNameFormat="FirstLetter" Height="200px" CellPadding="1"
    Font-Size="8pt" Font-Families="Verdana">
```

```

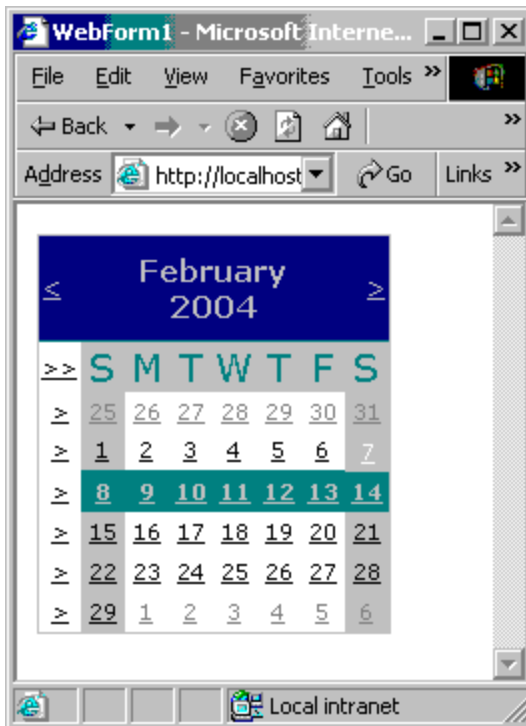
<TodayDayStyle BorderWidth="1px" ForeColor="White"
    BackColor="#99CCCC"></TodayDayStyle>
<NextPrevStyle Font-Size="8pt"
    ForeColor="#CCFF99"></NextPrevStyle>
<DayHeaderStyle Font-Size="Small" Font-Names="Verdana"
    Height="1px" ForeColor="#336666"
    BackColor="#99CCCC"></DayHeaderStyle>
<SelectedDayStyle Font-Bold="True" ForeColor="#CCFF99"
    BackColor="#009999"></SelectedDayStyle>
<TitleStyle Font-Size="11pt" Font-Bold="True" BorderWidth="1px"
    ForeColor="#CCFF99" BorderStyle="Solid" BorderColor="#3366CC"
    Width="221px" BackColor="#003399"></TitleStyle>
<WeekendDayStyle BackColor="#CCCCFF"></WeekendDayStyle>
<OtherMonthDayStyle ForeColor="#999999"></OtherMonthDayStyle>
</asp:calendar></p>
</form>

```

Hình 8-37 là kết quả cấu hình kể trên.

### 8.17.2.2 Làm việc với AdRotator

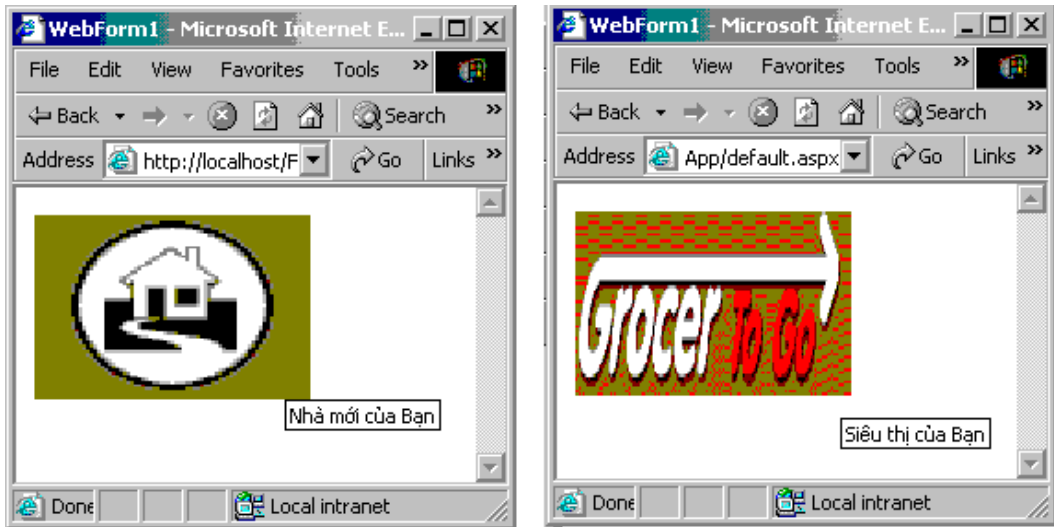
Mặc dù ASP cổ điển đã cung cấp một AdRotator (tắt chữ Advertisement Rotator, một loại quảng cáo quay vòng, giống như trên sân vận động bóng đá) nhưng qua ASP.NET thì nó được nâng cấp một cách đáng kể. Vai trò của ô control này là cho hiển thị một cách random những mẫu quảng cáo trên một vài vị trí trên browser. Khi bạn đặt nằm một ô control AdRotator trên khung thiết kế, thì đây mới là việc giữ trước chỗ. Về mặt chức năng, ô control này chỉ thật sự hoạt động khi bạn cho đặt để thuộc tính AdvertisementFile chỉ về một tập tin XML mô tả mỗi mẫu quảng cáo.



Hình 8-37: Calendar với màu mè

Dạng thức của tập tin quảng cáo khá đơn giản. Đối với mỗi mẫu quảng cáo, bạn tạo một phần tử <Ad>. Tối thiểu, mỗi phần tử <Ad> cho biết hình ảnh phải hiển thị (ImageUrl), URL phải lái về nếu hình ảnh được chọn ra (TargetUrl), phần văn bản mouseover (khi con trỏ nằm đè lên mẫu quảng cáo) và độ nặng đối với mẫu quảng cáo (Impressions). Thí dụ, giả sử

bạn có một tập tin (ads.xml) cho định nghĩa hai mẫu quảng cáo, như sau (hình 8-38):



Hình 8-38: Hai mẫu quảng cáo khác nhau...

```
<Advertisements>
<Ad>
  <ImageUrl>home.GIF</ImageUrl>
  <TargetUrl >http://www.Cars.com/home.GIF</TargetUrl >
  <AlternateText>Nhà mới của bạn</AlternateText>
  <Keyword>en-US</Keyword>
  <Impressions>80</Impressions>
</Ad>

<Ad>
  <ImageUrl>logo.GIF</ImageUrl>
  <TargetUrl >http://www.Cars.com/logo.GIF</TargetUrl >
  <AlternateText>Siêu thị của bạn</AlternateText>
  <Keyword>en</Keyword>
  <Impressions>80</Impressions>
</Ad>
</Advertisements>
```

Một khi bạn đã đặt để đúng đắn thuộc tính AdvertisementFile (và bảo đảm là các tập tin hình ảnh và XML đã vào đúng thư mục ảo (Cars) của ứng dụng Web, thì một trong hai mẫu quảng cáo (một siêu thị địa ốc, một siêu thị thực phẩm) sẽ được hiển thị một cách random khi người sử dụng lái về web site, như sau:

```
<form method="post" runat="server">
  <asp:AdRotator ID=AdRotator1 Runat="server" Width="150"
    Height="100" AdvertisementFile="ads.xml">
  </asp:AdRotator>
```

&lt;/form&gt;

Trong thí dụ trên các thuộc tính Width và Height của AdRotator được dùng để thiết lập kích thước của mẫu quảng cáo, mỗi mẫu mang kích thước 150x100px. Nếu mẫu nhỏ thua hoặc lớn hơn kích thước của AdRotator, thì bạn sẽ thấy các hình sẽ méo mó ít nhiều.

### 8.17.3 Các ô control chuyên về căn cứ dữ liệu (Datacentric Control)

WebForm định nghĩa một số ô control cho kết sinh HTML dựa trên (một phần) việc kết nối về một dữ liệu nguồn (data source). Như bạn có thể đoán được, các ô control này có thể được “nuôi dưỡng” bởi DataSet của ADO.NET, giống như các “song sinh” ở Windows Forms. Bảng 8-14 cho liệt kê vài WebForm Data Controls chủ chốt:

**Bảng 8-14: WebForm Data Control.**

WebForm Data Control	Mô tả
<b>DataGrid</b>	Một ô control cho hiển thị DataSet của ADO.NET trong một khung lưới.
<b>DataList</b>	Một ô control được gắn kết với một dữ liệu nguồn.

Ngoài những ô control WebForm chuyên về căn cứ dữ liệu, bạn nên nhớ là các ô control bẩm sinh có thể được cấu hình hóa hiển thị thông tin được lấy ra từ căn cứ dữ liệu hoặc từ UDT (user-defined type).

#### 8.17.3.1 Cho điền dữ liệu vào một DataGrid

Một trong những công tác thường xuyên trong việc triển khai phần mềm Web là đọc một dữ liệu nguồn để lấy thông tin rồi trả về dữ liệu dưới dạng một datagrid (dạng hàng/cột, tabular format). Khi sử dụng ASP cổ điển ta phải thiết lập một ADO RecordSet, xây dựng một bảng dữ liệu HTML on the fly sử dụng những tag HTML khác nhau. Với WebForm DataGrid ta cũng đi đến kết quả như trên nhưng ít rắc rối.

Giả sử bạn có một căn cứ dữ liệu mang tên Cars (kiểu \*.mdb), và khi một người sử dụng lại trang Web \*.aspx nào đó, bạn muốn họ đọc dữ liệu trên căn cứ dữ liệu Cars. Nhiệm vụ đầu tiên của bạn là viết một hàm thụ lý tình huống **Page\_Load()**. Một khi làm xong, bạn mới có thể sau đó tạo một đối tượng **DataSet** và gắn kết trực tiếp đối tượng này vào một **DataGrid**. Sau đây là đoạn mã C#:

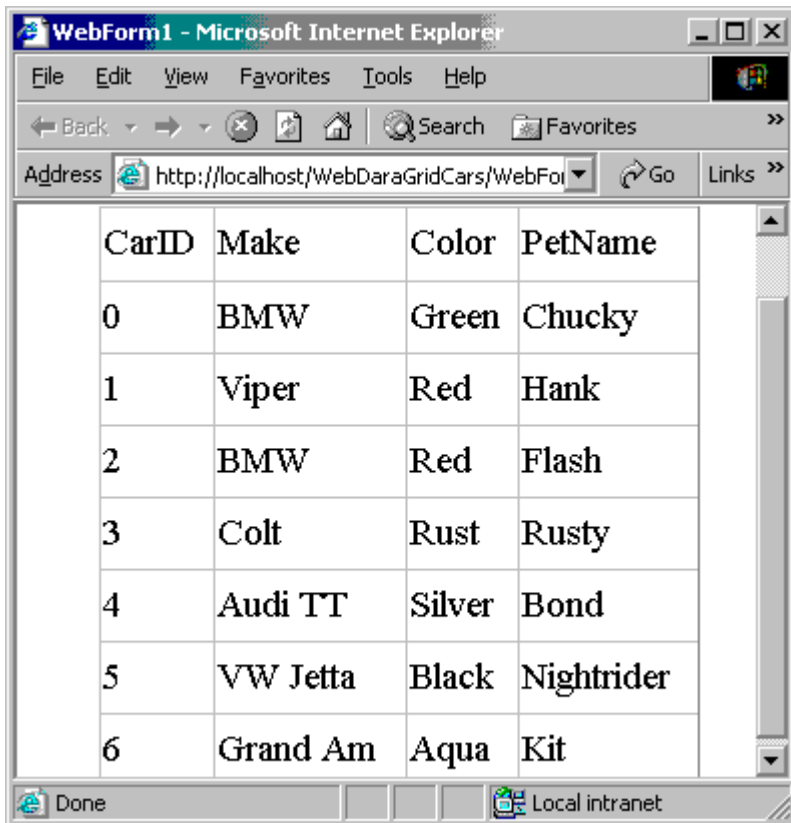
```
private void Page_Load(object sender, System.EventArgs e)
{
    if(!IsPostBack)
    {
        OleDbConnection conn = new OleDbConnection();
```

```

conn.ConnectionString = "Provider=Microsoft.Jet.OLEDB.4.0;"+
    "Data Source=D:\\Thien\\Cars.mdb";
OleDbDataAdapter da = new OleDbDataAdapter(
    "Select * from Inventory", conn);
DataSet ds = new DataSet();
da.Fill(ds, "Inventory");
DataGrid1.DataSource = ds.Tables["Inventory"].DefaultView;
DataGrid1.DataBind();
    }
}

```

Bạn thấy là căn cứ dữ liệu Cars.mdb (dạng thức Microsoft Access) chúng tôi hiện cho về thư mục D:\Thien. Nếu bạn chưa quen với ADO.NET thì đề nghị bạn tìm xem tập IV của bộ sách .NET toàn tập này. Hình 8-39 cho thấy kết xuất.



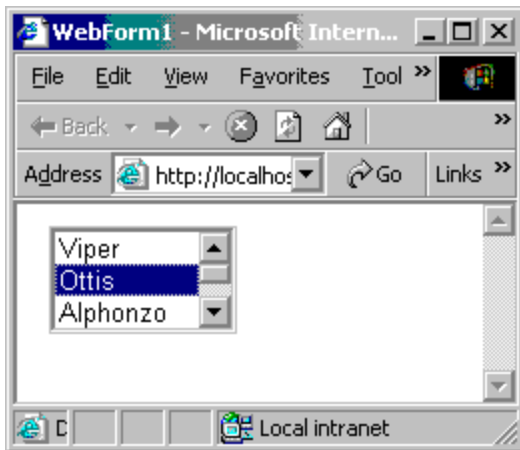
Hình 8-39: Sử dụng DataAdapter để ãienn DataGrid Web

### 8.17.3.2 Đôi điều về Data Binding

Như bạn có thể thấy, **DataGrid** control đem lại cho các thành viên **DataSource** và **DataBind()** khả năng hiển thị nội dung của một **DataTable** nào đó. Đây là một bước làm bùng nổ việc triển khai các hệ thống trong xí nghiệp. Tuy nhiên, WebForm control (cũng như đối với Windows Forms control) còn cho phép bạn gắn kết (bind) với các dữ liệu nguồn khác trên một ô control nào đó.

Thí dụ, giả sử bạn có một nhóm trị được biểu diễn bởi một bản dãy đơn giản. Bạn có thể dùng cùng kỹ thuật như với việc gắn kết về kiểu dữ liệu **DataGrid**. Bạn có thể gắn liền một bản dãy về một kiểu dữ liệu GUI. Thí dụ, nếu bạn có một ô ASP.NET ListBox control (với mã nhận diện ID là petNameList) trên trang Web \*.aspx, bạn có thể nhậ tu hàm thụ lý tình huống Page\_Load như sau:

```
private void Page_Load(object sender, System.EventArgs e)
{
    if(!IsPostBack)
    {
        // Tạo một bản dãy dữ liệu cho gắn liền với listbox
        string[] carPetNames= {"Viper", "Hank", "Ottis", "Alphonso",
                               "Cage", "TB"};
        petNameList.DataSource = carPetNames;
        petNameList.DataBind();
    }
}
```



**Hình 8-40: Gắn kết dữ liệu với ô control thông dụng WebForm.**

Như bạn có thể thấy hình 8-40 là kết xuất của đoạn mã kể trên. Bạn nhớ lại tất cả các bản dãy .NET đều được ánh xạ lên kiểu dữ liệu System.Array. Sự kiện là bất cứ kiểu dữ liệu nào thì công giao diện IEnumerable có thể được gắn kết vào một ô control GUI. Do đó, nếu bạn nhậ tu chuỗi bản dãy đơn giản thành một thể hiện của kiểu dữ liệu ArrayList, thì kết xuất cũng y chang. Thí dụ, ta viết như sau:

```
private void Page_Load(object sender,
                        System.EventArgs e)
{
    if(!IsPostBack)
    {
        // Bây giờ bạn dùng một
        // ArrayList
        ArrayList carPetNames = new
                                ArrayList();
```

```
        carPetNames.Add("Viper");
        carPetNames.Add("Ottis");
        carPetNames.Add("Alphonzo");
        carPetNames.Add("Cage");
```

```

        carPetNames.Add("TB");
        petNameList.DataSource = carPetNames;
        petNameList.DataBind();
    }
}

```

Hình 8-40 cũng là kết xuất của đoạn mã trên.

## 8.17.4 Các ô control kiểm tra hợp lệ dữ liệu (Validation Controls)

Khái niệm cuối cùng được gán cho các ô control WebForm là kiểm tra hợp lệ dữ liệu do người sử dụng khó vào. Bảng 8-15 liệt kê các ô control validation cốt lõi.

**Bảng 8-15: Validation Controls**

WebForm Validation Control	Mô tả
<b>CompareValidator</b>	Kiểm tra hợp lệ xem trị của một input control có bằng với một trị nào đó trên một input control khác.
<b>CustomValidator</b>	Cho phép bạn xây dựng một hàm kiểm tra hợp lệ custom lo kiểm tra một ô control nào đó.
<b>RangeValidator</b>	Xác định một trị nào đó nằm trong một biên độ (range) đã định sẵn trước.
<b>RegularExpressionValidator</b>	Kiểm tra liệu xem trị của một input control có khớp với một pattern của một regular expression hay không.
<b>RequiredFieldValidator</b>	Bảo đảm là một input control nào đó chứa một trị (và như vậy không thể trống rỗng).
<b>ValidationSummary</b>	Cho hiển thị một tóm lược các sai lầm về kiểm tra hợp lệ của một trang trên một danh sách, bulleted list hoặc một paragraph đơn giản. Các sai lầm có thể được hiển thị inline và/hoặc trong một popup message box.

Để minh họa những điểm cơ bản làm việc với các ô control validation, ta thử tạo một dự án C# Web, cho mang tên **ValidateWebApp**. Sau đó cho đổi tập tin \*.aspx thành default.aspx rồi cho mở design time template. Tiếp theo, bạn tạo một giao diện người sử dụng đơn giản như theo hình 8-41 sử dụng kỹ thuật lỗi thả chuẩn. Bạn nên dùng tag HTML để tạo dòng văn bản Car News Letter Sign Up thay vì dùng Label.

Tiếp theo, ta thử xem sử dụng thế nào các ô control validation của WebForm. Để minh họa giả sử bạn muốn ô text box txtEmail phải chứa thông tin gì đó trước khi bạn submit biểu mẫu cho Web server. Vào lúc thiết kế, bạn yêu cầu ô control **RequiredFieldValidator** có mặt trên biểu mẫu. Nghĩa là bạn lôi thả ô control này lên biểu mẫu. Sau đó, bạn sử dụng cửa sổ **Properties**, bạn cho đặt để thuộc tính **ErrorMessage** về một trị gì đó



được hiển thị khi kiểm tra hợp lệ bất thành (trong trường hợp của chúng tôi là thông điệp “Ta cần địa chỉ e-mail”), cũng như thiết lập mã ID của ô control mà widget này sẽ thực hiện việc kiểm tra hợp lệ bằng cách dùng thuộc tính **ControlToValidate** (ở đây là txtEMail). Xem hình 8-42.



Hình 8-41: Một UI đơn giản

Sau hậu trường, \*.aspx logic có thể là như sau:

```
<asp:RequiredFieldValidator id=
  "RequiredFieldValidator1"
  style="Z-INDEX: 108; LEFT: 263px;
  POSITION: absolute; TOP: 201px"
  runat="server" Height="41px" Width="125px"
  ErrorMessage="Ta cần địa chỉ e-mail"
  ControlToValidate="txtEMail">
</asp:RequiredFieldValidator>
```

Ngoài ra, bạn sẽ có một lớp được dẫn xuất từ Page (như đã được khai báo bởi attribute Codebehind) như là một biến thành viên mới như sau:

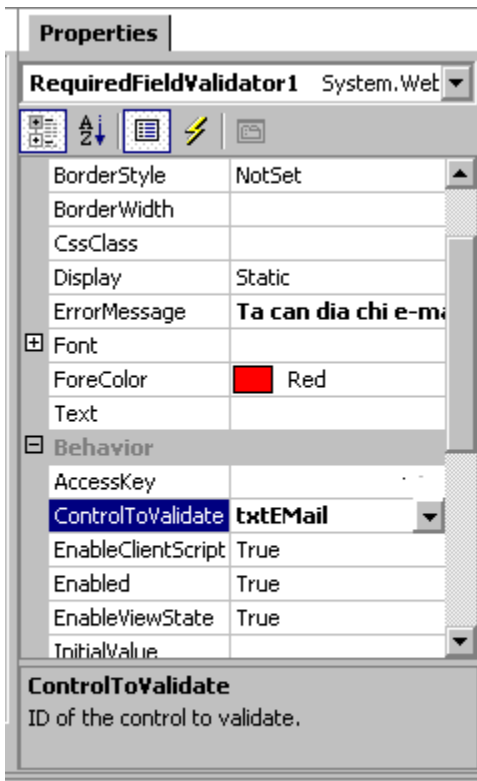
```
public class WebForm1: System.Web.UI.Page
{
    protected System.Web.UI.WebControls.TextBox TextBox1;
```

```

protected System.Web.UI.WebControls.TextBox TextBox2;
protected System.Web.UI.WebControls.Label Label1;
protected System.Web.UI.WebControls.Label Label2;
protected System.Web.UI.WebControls.Label Label3;
protected System.Web.UI.WebControls.RequiredFieldValidator
    RequiredFieldValidator1;
protected System.Web.UI.WebControls.TextBox txtEmail;
protected System.Web.UI.WebControls.Button Button1;

...
}

```



Hình 8-42: Cấu hình hoá việc kiểm tra hợp lệ

Bây giờ bạn cho Build và cho Start. Khi trang Web hiện lên bạn không thấy gì lạ. Bạn khở tên vào ô First Name rồi ấn nút <Submit>, chưa điền gì vào ô textbox e-mail. Lúc này thông điệp sai lầm màu đỏ hiện lên như theo hình 8-43:

Một khi bạn khở gì đó vào ô text box e-mail thì thông điệp sai lầm trên sẽ biến khỏi. Nếu bạn xem HTML hiển thị trên browser, bạn sẽ thấy hàm RequiredFieldValidator được kết sinh viết theo JavaScript, và bạn khỏ nhọc công viết. Tuy nhiên, nếu validation control xác định là browser nào trình yêu cầu HTTP không hỗ trợ logic như thế, thì trang HTML được trả về sẽ chuyển hướng việc xử lý thông điệp sai lầm về Web server.

## 8.17.5 Thụ lý tình huống WebForms Control

Như bạn có thể thấy, một cách tiếp cận phổ biến là thụ lý các tình huống phát pháo bởi các ô control HTML GUI là thông qua sử dụng JavaScript phía client. Trong bất cứ trường hợp nào mà việc thụ lý tình huống đòi hỏi bất cứ logic hiển thị, báo động browser (nghĩa là message box) hoặc các tương tác trực tiếp khác với mô hình đối tượng của browser, thì đây là cách tiến hành. Tuy nhiên, đối với các trường hợp khác bạn lại có

những ô control ASP.NET thực hiện những việc xử lý không mang tính GUI (như tiến hành tính toán, hiệu đính một bảng dữ liệu, v.v..).



Hình 8-43: RequiredFieldValidator hoạt động tốt

Mặc dù bạn vẫn tự do sử dụng ngôn ngữ kịch bản phía client cho những mục đích này, ASP.NET không cung cấp của bạn một phương án thay thế khác. Mỗi ô control WebForm sẽ phản ứng đối với tập hợp những tình huống riêng của nó mà ta có thể cấu hình hóa thụ lý các hàm tình huống *trên phía server*. Muốn cấu hình hóa một ô control làm việc này, bạn chỉ cần thêm một hàm thụ lý tình huống sử dụng của sổ Properties.

Thí dụ, giả sử bạn cần ấn định ngày nào được chọn trên ô control Calendar. Bạn có thể chặn hứng tình huống SelectionChanged và hành xử một cách thích ứng, như sau:

```
protected void Calendar1_SelectionChanged(  
    object sender, System.EventArgs e)  
{  
    Response.Write("<h5>Xe của bạn sẽ được giao vào ngày: " +  
        Calendar1.SelectedDate.Date + "</h5>");  
}
```

Bây giờ, khi người sử dụng kích hoạt tình huống **SelectionChanged**, browser sẽ thi hành một post back về server để triệu gọi đúng hàm thụ lý tình huống.

Mặc dù ASP.NET cung cấp khả năng thụ lý các tình huống của các ô control GUI trên Web server, bạn nên để ý đến những bất cập hiển nhiên. Việc này sẽ tăng số postback đối với máy tính nằm ở xa, do đó sẽ giảm hiệu năng ứng dụng Web của bạn. Tuy nhiên, khi sử dụng một cách cẩn thận những hàm thụ lý tình huống phía server sẽ dẫn đến việc duy trì những đoạn mã modular và khả tín hơn.

## 8.18 Một ứng dụng Web Form

Bây giờ ta thử tạo một thí dụ trọn vẹn về Web Forms. Bạn cho khởi động Visual Studio.NET, ra lệnh **File | Project** rồi chọn **Visual C# project** và **ASP.NET Web Application**, và đặt tên dự án là **ProgrammingCSharpWeb**. Visual Studio.NET sẽ cho hiển thị http://localhost như là thư mục ảo. Khi bạn ấn OK thì Visual Studio.NET sẽ tạo các tập tin hình thành dự án.

Visual Studio.NET đưa tất cả các tập tin dự án nó tạo ra trên một thư mục nằm trong lòng web site mặc nhiên trên máy cục bộ của bạn. Thí dụ, **C:\Inetpub\wwwroot\ProgrammingCSharpWeb**. Bạn dùng Window Explorer thì thấy rõ.

Khi ứng dụng được tạo ra, Visual Studio.NET đưa một số tập tin vào dự án. Bản thân Web Form được trữ trong một tập tin mang tên **WebForm1.aspx**. Tập tin này chỉ chứa toàn HTML. Một tập tin thứ hai, không kém phần quan trọng, WebForm1.cs sẽ lo trữ đoạn mã C# được gắn liền với biểu mẫu; đây là một tập tin được gọi là code-behind.file.

Trên Solution Explorer, tập tin code-behind không xuất hiện. Muốn xem nội dung tập tin này (.cs), bạn chỉ cần ở trên biểu mẫu, right click lên biểu mẫu, rồi chọn click mục View Code trên trình đơn shortcut, thì nội dung WebForm1.aspx.cs sẽ hiện lên. Bây giờ bạn thấy có hai tab: WebForm1.aspx và WebForm1.aspx.cs. Bạn chỉ cần click qua lại giữa hai Tab này để làm việc khi thiết kế giai đoạn khi thêm đoạn mã. Khi đang ở thiết kế (nghĩa là Tab WebForm1.aspx) bạn có thể chọn giữa **Design** mode và **HTML** mode bằng cách click Tab nằm ở cuối màn hình Editor window. Design mode cho phép bạn lôi thả các ô control lên biểu mẫu, còn HTML mode cho phép bạn view và hiệu đính trực tiếp đoạn mã HTML.

Bạn vừa tạo một Web Form trống rỗng. Điều đầu tiên là bạn thêm một dòng văn bản gì đó lên biểu mẫu. Thí dụ, “Anh em SAMIS xin gửi lời chào!” ... Thí dụ, bạn viết sau tag <Form>:

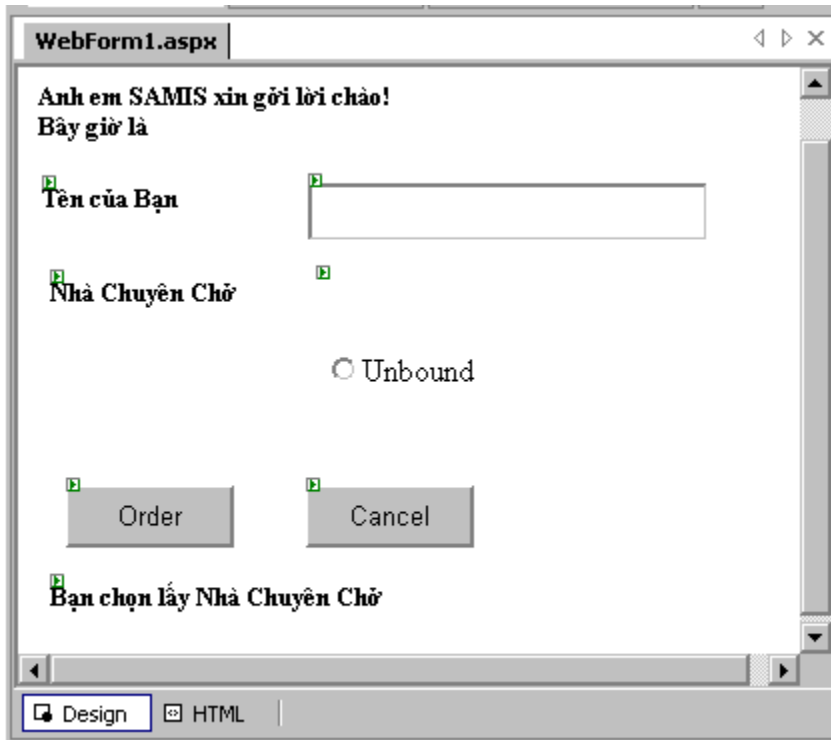
```
<h5>
```

```

<b>Anh em SAMIS xin gửi lời chào!</b>
<b>Bây giờ là <% Response.Write(DateTime.Now.ToString());%></b>
</h5>

```

Sau lời chào mừng là ngày tháng năm, sử dụng lớp DateTime. Sau đó bạn dùng Toolbox lôi thả một vài ô control tạo thành giao diện như sau (hình 8-44):



Hình 8-44: Giao diện vào lúc thiết kế

Bạn có thể thấy giao diện này gồm 3 label, một ô text box, 1 radio button list, và hai button. Khi bạn lôi các ô control này từ toolbox lên thả trên web form, rồi bạn chỉnh các thuộc tính Text và (ID) thì Visual Studio.NET kết sinh ra những lệnh như sau:

```

protected System.Web.UI.WebControls.Label Label1;
protected System.Web.UI.WebControls.Label Label2;
protected System.Web.UI.WebControls.Button Order;
protected System.Web.UI.WebControls.Button Cancel;
protected System.Web.UI.WebControls.RadioButtonList rbl1;
protected System.Web.UI.WebControls.TextBox txtName;

```

Đặc biệt bạn muốn tạo những radio button dựa trên dữ liệu trong một căn cứ dữ liệu vì bạn không biết vào lúc thiết kế sẽ có những radio button nào, và bao nhiêu nút bạn cần

đến. Muốn thực hiện được điều này, bạn sử dụng ô control **RadioButtonList**. Lệnh sau đây định nghĩa một **RadioButtonList**:

```
protected System.Web.UI.WebControls.RadioButtonList rbl1;
```

**RadioButtonList** là một ô control cho phép bạn tạo những radio button theo lập trình. Bạn chỉ cần cung cấp tên và trị đối với các nút này, và ASP.NET sẽ lo việc “xào nẫu” còn lại. Bạn xóa đi những radio button đã nằm trên biểu mẫu rồi lôi thả một **RadioButtonList** vào thế chỗ. Một khi nó đã hiện diện, bạn dùng cửa sổ Properties đổi lại tên thành **rbl1**.

Tình huống quan trọng nhất là `Page_Load`, được phát pháo mỗi lần biểu mẫu Web được nạp vào. Khi trang Web được nạp vào bạn muốn điền các radio button với những trị lấy từ căn cứ dữ liệu xuống. Thí dụ, bạn đang tạo một biểu mẫu đặt hàng, bạn muốn tạo một radio button cho mỗi loại phương tiện chuyên chở: UPS, FedEx, v.v.. Do đó, trong tình huống `Page_Load` bạn phải đưa đoạn mã tạo những radio button này dựa vào trị lấy từ căn cứ dữ liệu xuống.

Bạn chỉ muốn nạp những trị này vào radio button lần đầu tiên khi trang web được nạp vào. Nếu người sử dụng click một nút hoặc tiến hành một hành động nào gọi trang lại cho server, thì bạn không muốn tìm lại lần nữa các trị khi trang được nạp lại.

ASP.NET có thể phân biệt lần đầu tiên khi trang web được hiển thị lần đầu tiên so với những lần sau sau khi khách hàng postback trang web cho server. Mỗi trang Web Form đều có thuộc tính `IsPostBack`, và thuộc tính này là true nếu trang được nạp đáp ứng đối với postback của khách hàng và false nếu nó được nạp vào lần đầu tiên.

Bạn có thể kiểm tra trị của `IsPostBack`. Nếu nó false, thì bạn biết rằng đây là lần đầu tiên trang được nạp vào, và đây là lúc thuận tiện đi lấy dữ liệu từ căn cứ dữ liệu xuống:

```
protected void Page_Load(object sender, EventArgs e)
{
    if (!IsPostBack)
    {
        //...
    }
}
```

## 8.18.1 Kết nối với căn cứ dữ liệu

Đoạn mã dùng kết nối với một căn cứ dữ liệu cũng như điền dữ liệu vào một dataset có thể bạn chưa quen lắm, nhưng tạm thời bạn chấp nhận những gì chúng tôi giải thích sau đây. Về sau, bạn có thể tham khảo tập IV bộ sách này khi đề cập đến ADO.NET khi chúng tôi đi sâu vào căn cứ dữ liệu.

Trong thí dụ này, chúng tôi dùng công nghệ OleDb (thay vì SQL) đối với căn cứ dữ liệu Northwind.mdb mà Microsoft cung cấp kèm theo SDK. Trong căn cứ dữ liệu Northwind bạn sẽ dùng đến bảng dữ liệu Shippers (nhà chuyên chở) để điền dữ liệu vào radio button.

Bạn bắt đầu khai báo ở đầu dự án chỉ thị using đối với OleDb:

```
using System.Data.OleDb;
```

và các biến thành viên mà bạn cần đến:

```
private System.Data.OleDb.OleDbConnection conn;  
private System.Data.OleDb.OleDbDataAdapter dad;  
private System.Data.DataSet dst;  
private System.Data.OleDb.OleDbCommand cmd;
```

Bạn tạo một đối tượng OleDbConnection rồi cho mở kết nối này. Để cho dễ làm việc, chúng tôi cho sao một bản căn cứ dữ liệu Northwind ghi vào thư mục làm việc của chúng tôi D:\Thien. Chúng tôi tạo một connectionString đối với căn cứ dữ liệu Northwind và dùng biến này để thể hiện và mở một đối tượng OleDbConnection:

```
string strConn = @"Provider=Microsoft.Jet.OLEDB.4.0;" +  
                  "Data Source=D:\\Thien\\Northwind.mdb";  
conn = new OleDbConnection(strConn);  
conn.Open();
```

Tiếp theo, bạn cho tạo một đối tượng DataSet và cho xử lý các truy vấn case-sensitive:

```
dst = new DataSet();  
dst.CaseSensitive = true;
```

Kế tiếp là tạo một đối tượng **OleDbCommand** và gán cho nó đối tượng connection và chỉ thị **Select**, cần thiết đi lấy **ShipperID** (mã nhận diện nhà chuyên chở) và tên công ty nhận diện mỗi nhà chuyên chở. Ta dùng tên như là văn bản đối với radio button và **ShipperID** như là trị:

```
cmd = new OleDb.OleDbCommand();  
cmd.Connection = conn;  
cmd.CommandText = "Select ShipperID, CompanyName From Shippers";
```

Bây giờ bạn tạo một đối tượng OleDbDataAdapter, cho đặt để thuộc tính SelectCommand của đối tượng này và thêm bảng dữ liệu Shippers vào bảng ánh xạ các bảng dữ liệu:

```
dad = new OleDbDataAdapter();  
dad.SelectCommand = cmd;
```

```
dad.TableMappings.Add("Table", "Shippers");
```

Cuối cùng, bạn cho điền đối tượng dad với kết quả của truy vấn:

```
dad.Fill(dst);
```

Coi như tới đây, nếu may mắn đối tượng DataSet, dst, đã điền đầy dữ liệu của bảng dữ liệu Shippers.

Ta sẽ cho gắn kết **dst** với đối tượng **RadioButtonList**, rbl1 mà ta đã tạo ra trước đó. Thuộc tính đầu tiên **RepeatLayout** báo cho **RadioButtonList** biết cách bố trí các radio button thế nào lên biểu mẫu:

```
rbl1.RepeatLayout = System.Web.UI.WebControls.RepeatLayout.Flow;
```

**Flow** là một trong những trị của enumeration **RepeatLayout**. Trị khác là **Table** cho bố trí radio button theo kiểu hàng/cột (tabular). Tiếp theo, bạn phải báo cho **RadioButtonList** biết những trị nào từ **DataSet** sẽ được dùng để hiển thị (**DataField**) và trị nào phải trả về khi được chọn bởi người sử dụng (**DataValueField**):

```
rbl1.DataTextField = "CompanyName";  
rbl1.DataValueField = "ShipperID";
```

Bước cuối cùng là báo cho RadioButtonList loại view nào sẽ sử dụng dữ liệu. Đối với thí dụ này, bạn sẽ dùng view mặc nhiên, DefaultView, của bảng dữ liệu Shippers trong lòng data set:

```
rbl1.DataSource = dst.Tables["Shippers"].DefaultView;
```

Bây giờ bạn sẵn sàng gắn kết RadioButtonList với dataset:

```
rbl1.DataBind();
```

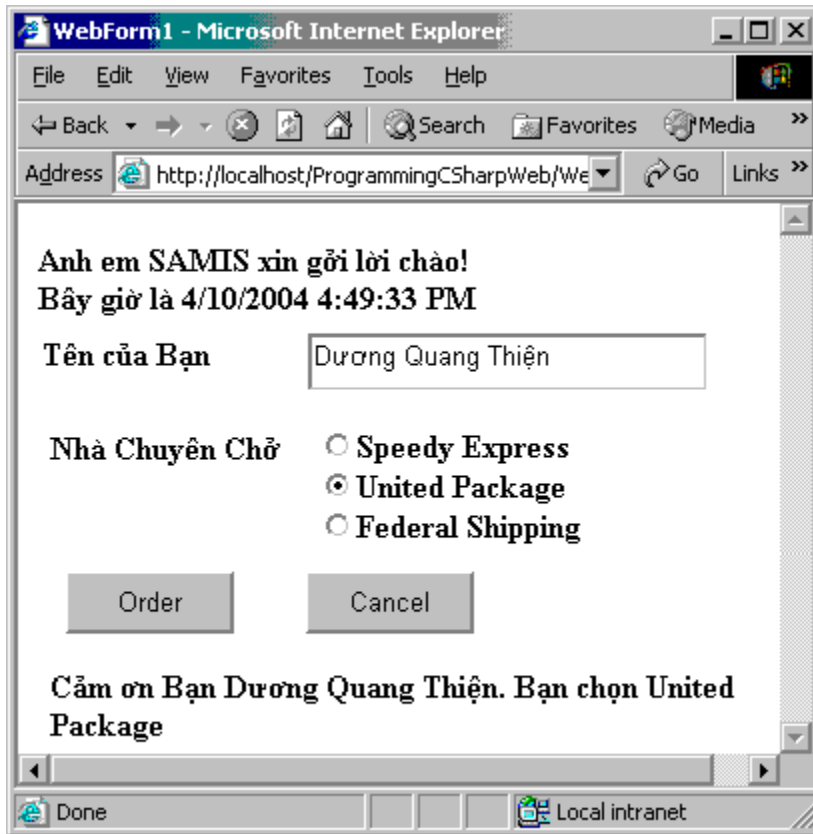
Cuối cùng, bạn phải bảo đảm một trong những radio button được chọn; ta chọn nút đầu tiên:

```
rbl1.Items[0].Selected = true;
```

Lệnh trên truy cập vào collection **Items** trong lòng **RadioButtonList**, chọn ra mục đầu tiên và cho thuộc tính **Selected** về true.

Khi bạn cho chạy chương trình thì các nút radio button sẽ được hiển thị như theo hình 8-45. Bạn so sánh RadioButtonList trên biểu mẫu thiết kế (hình 8-44) với những nút được hiển thị trên hình 8-45.





Hình 8-45: Page posted sau khi người sử dụng click Order

## 8.18.2 Thụ lý tình huống Postback

Các đối tượng `<asp:button>` tự động postback khi bị click. Bạn khỏi viết bất cứ đoạn mã nào đối với tình huống này trừ khi bạn muốn làm gì đó hơn là postback đối với server. Nếu bạn không có hành động gì khác, thì trang web sẽ được gọi trả lại cho khách hàng.

Thông thường, khi một trang web được tô vẽ lại, thì mỗi ô control sẽ được vẽ lại từ đầu từ số không. Web thuộc loại stateless, cho nên nếu bạn muốn quản lý tình trạng của mỗi ô control, bạn phải tự làm lấy. Trên ASP cổ điển, lập trình viên chịu trách nhiệm quản lý tình trạng này, nhưng với ASP.NET thì bạn được hỗ trợ, vì khi trang web được post, một thuộc tính ẩn dấu ViewState được thêm vào trang. Thuộc tính này tượng trưng cho tình trạng của biểu mẫu. Khi phải vẽ lại trang web, ASP.NET sẽ dùng thuộc tính này để cho hoàn nguyên các ô control về tình trạng trước kia.

Khi người sử dụng click lên nút <Order>, thì trang web sẽ được posted và hàm thụ lý tình huống Click của nút này sẽ được triệu gọi:

```
private void Order_Click(object sender, System.EventArgs e)
{
    string msg;
    msg = "Cảm ơn bạn " + txtName.Text + ". u66 ?ạn chọn ";
    for (int i = 0; i < rbl1.Items.Count; i++)
    {
        if(rbl1.Items[i].Selected)
        {
            msg = msg + rbl1.Items[i].Text;
            lblFeedBack.Text = msg;
        }
    }
}
```

Chắc bạn hiểu ý nghĩa của đoạn mã này.

Bạn cho chạy chương trình, và khi biểu mẫu hiện lên, bạn cho điền tên bạn, chọn nút United Package Service, rồi ấn nút <Order> thì biểu mẫu sẽ được trình duyệt (submit) và rồi được hiển thị lại. Kết quả là hình 8-45. Bạn so sánh nội dung của lblFeedBack nằm ở cuối biểu mẫu hình 8-44 với 8-45, bạn thấy nội dung label được nhật tu một cách thích ứng.

Sau đây là toàn bộ liệt in dự án ProgrammingCSharpWeb.

```
using System;
using System.Collections;
using System.ComponentModel;
using System.Data;
using System.Drawing;
using System.Web;
using System.Web.SessionState;
using System.Web.UI;
using System.Web.UI.WebControls;
using System.Web.UI.HtmlControls;
using System.Data.OleDb;

namespace ProgrammingCSharpWeb
{
    public class WebForm1: System.Web.UI.Page
    {
        protected System.Web.UI.WebControls.Label Label1;
        protected System.Web.UI.WebControls.Label Label2;
        protected System.Web.UI.WebControls.Button Order;
        protected System.Web.UI.WebControls.Button Cancel;
        protected System.Web.UI.WebControls.RadioButtonList rbl1;
        protected System.Web.UI.WebControls.TextBox txtName;
        protected System.Web.UI.WebControls.Label lblFeedBack;
        private System.Data.OleDb.OleDbConnection conn;
        private System.Data.OleDb.OleDbDataAdapter dad;
        private System.Data.DataSet dst;
        private System.Data.OleDb.OleDbCommand cmd;
```

```

private void Page_Load(object sender, System.EventArgs e)
{
    if(!IsPostBack)
    {
        string strConn = @"Provider=Microsoft.Jet.OLEDB.4.0;" +
            "Data Source=D:\\Thien\\Northwind.mdb";
        conn = new OleDbConnection(strConn);
        conn.Open();
        dst = new DataSet();
        dst.CaseSensitive = true;
        cmd = new OleDb.OleDbCommand();
        cmd.Connection = conn;
        cmd.CommandText = "Select ShipperID, CompanyName From Shippers";
        dad = new OleDbDataAdapter();
        dad.SelectCommand = cmd;
        dad.TableMappings.Add("Table", "Shippers");
        dad.Fill(dst);
        rb11.RepeatLayout = RepeatLayout.Flow;
        rb11.DataTextField = "CompanyName";
        rb11.DataValueField = "ShipperID";
        rb11.DataSource = dst.Tables["Shippers"].DefaultView;
        rb11.DataBind();
        rb11.Items[0].Selected = true;
    }
}

override protected void OnInit(EventArgs e)
{
    InitializeComponent();
    base.OnInit(e);
}

private void InitializeComponent()
{
    this.Order.Click += new System.EventHandler(this.Order_Click);
    this.Load += new System.EventHandler(this.Page_Load);
}

private void Order_Click(object sender, System.EventArgs e)
{
    string msg;
    msg = "Cảm ơn bạn " + txtName.Text + ". Bạn chọn ";
    for (int i = 0; i < rb11.Items.Count; i++)
    {
        if(rb11.Items[i].Selected)
        {
            msg = msg + rb11.Items[i].Text;
            lblFeedBack.Text = msg;
        }
    }
}
}
}
}

```

Trong chương này, chúng tôi mới phác họa sơ về ASP.NET và căn cứ dữ liệu. Muốn tìm hiểu sâu về các lĩnh vực này, bạn có thể tìm đọc tập IV bộ sách .NET toàn tập này để cập đến căn cứ dữ liệu và ADO.NET và tập V bộ sách này để cập đến lập trình Web với ASP.NET.

## Chương 9

# Lập trình Web Services

.NET Web Services nói rộng khái niệm xử lý phân tán (distributed processing) dùng xây dựng những cấu kiện (component) theo đây các hàm hành sự cấu kiện có thể được triệu gọi thông qua Internet. Các cấu kiện này có thể được xây dựng sử dụng bất cứ ngôn ngữ nào chịu chơi .NET, và các cấu kiện này có thể liên lạc sử dụng những nghi thức (protocol) mở mang tính độc lập đối với sàn diễn (platform-independent). Trong chương này bạn sẽ xem xét và tiêu thụ các ASP.NET Web services. Nói một cách đơn giản, một Web service là một đoạn mã managed đơn vị (điển hình được cài đặt dưới IIS) mà ta có thể triệu gọi từ xa sử dụng HTTP.

Như bạn có thể thấy, Web Services bao gồm 3 công nghệ hỗ trợ: một ngôn ngữ mô tả được gọi là WSDL (Web Service Description Language), một nghi thức truyền tin (wire protocol, chẳng hạn HTTP GET, HTTP POST, hoặc SOAP), và một dịch vụ phát hiện (discovery service, được thể hiện bởi các tập tin \*.vsdisco). Trong chương này, bạn sẽ bắt đầu tạo một **Calculator Web Service**, rồi từ đó tạo một trung tâm xe hơi trả về DataSet của ADO.NET, ArrayList và các kiểu dữ liệu custom khác.

Một khi bạn đã trưng ra các khối xây dựng cốt lõi của .NET Services, chương này sẽ kết thúc bằng cách cho thấy làm thế nào xây dựng một lớp proxy (sử dụng Visual Studio .NET cũng như trình tiện ích **wsdl.exe**) có thể được tiêu thụ bởi những ứng dụng client Web-based, console-based, và Windows Forms.

## 9.1 Tìm hiểu vai trò của Web Services

Nhìn từ trên cao, ta có thể định nghĩa một Web Service như là một đoạn mã đơn vị mà ta có thể cho hiện dịch (activate) sử dụng các HTTP request. Bây giờ, ta thử nhìn lại quá khứ một chút. Trước kia, truy cập từ xa các đoạn mã nhị phân đòi hỏi những nghi thức thuộc sàn diễn đặc trưng (và thỉnh thoảng một ngôn ngữ đặc trưng). Một thí dụ cổ điển về cách tiếp cận này là DCOM (tắt chữ Distributed COM). DCOM client truy cập các kiểu dữ liệu COM từ xa sử dụng các triệu gọi RPC (Remote Procedure Call) được gắn kết chặt chẽ. CORBA cũng đòi hỏi các nghi thức được gắn kết chặt chẽ để hiện dịch những kiểu dữ liệu từ xa. EJB (Enterprise Java Beans) đòi hỏi một nghi thức đặc trưng và một ngôn ngữ đặc trưng (Java). Vấn đề ở đây là các kiến trúc xử lý từ xa này là phải có

những nghi thức truy cập mang tính đặc hữu (proprietary), đòi hỏi kết nối chặt chẽ với dữ liệu nguồn nằm ở xa.

Như bạn đã biết, sử dụng C#, VB .NET hoặc bất cứ ngôn ngữ nào chịu chơi .NET, bạn có thể xây dựng những kiểu dữ liệu có thể đem đi tiêu thụ và nói rộng xuyên qua ranh giới ngôn ngữ. Bằng cách sử dụng Web Service bạn có thể truy cập những assembly (trung lập đối với ngôn ngữ) chỉ sử dụng HTTP mà thôi. Trong tất cả các nghi thức hiện có trên thị trường, HTTP là một nghi thức liên lạc đặc thù mà tất cả mọi sản phẩm đều chấp nhận hỗ trợ.

Như vậy, sử dụng Web Service, bạn (là người triển khai Web Service) có thể sử dụng bất cứ ngôn ngữ nào mình muốn. Còn bạn (là người tiêu thụ Web Service) có thể dùng HTTP chuẩn để triệu gọi các hàm hành sự trên những kiểu dữ liệu được định nghĩa trên Web Service. Điểm mấu chốt của vấn đề là thành linh bạn có một ngôn ngữ thật sự và một “hội nhập” sản phẩm. Không còn nói đến COM hoặc Java hoặc CORBA nữa. Chỉ còn nói đến HTTP và ngôn ngữ lập trình bạn ưa thích (lẽ dĩ nhiên là C#). Như bạn có thể thấy, SOAP (Simple Object Access Protocol) và XML (Extended Markup Language) cũng sẽ là hai thành phần chủ chốt trong kiến trúc của Web Service, được sử dụng phối hợp với HTTP chuẩn.

Giống như bất cứ assembly .NET nào, một Web Service thường chứa một số lớp, giao diện, structure, và enumeration cung cấp chức năng hộp đen đối với các client nằm xa. Hạn chế thật sự duy nhất mà bạn phải quan tâm là vì Web Service được thiết kế để cho việc triệu gọi từ xa dễ dàng nên bạn phải tránh sử dụng lô gic thiên về GUI. Điển hình là Web Service định nghĩa những đối tượng business thì hành một đơn vị công tác (nghĩa là tính toán, đọc dữ liệu nguồn, hoặc gì gì đó) đối với người tiêu thụ và chờ yêu cầu kế tiếp.

Một khía cạnh của Web Service mà có thể bạn không sẵn sàng hiểu thấu là việc người tiêu thụ Web Service không nhất thiết phải là client sử dụng browser. Như bạn có thể thấy, các ứng dụng console hoặc Windows Form đều có thể tiêu thụ Web Service một cách dễ dàng. Trong mỗi trường hợp, khách hàng tương tác gián tiếp với Web Service thông qua một proxy trung gian. Proxy được xem như là thật sự một kiểu dữ liệu nằm ở xa cho trung ra cùng tập hợp các thành viên. Tuy nhiên, ở sau hậu trường đoạn mã của proxy làm nhiệm vụ chuyển đến những yêu cầu cho Web Service sử dụng HTTP chuẩn hoặc tùy chọn các thông điệp SOAP.

## 9.2 Các thành phần cốt lõi của Web Service

Web Service thường điển hình được lưu trữ bởi IIS nằm dưới một thư mục ảo duy nhất, cũng giống như với một ASP.NET chuẩn. Tuy nhiên, ngoài managed code tạo

thành chức năng được xuất khẩu, một Web Service đòi hỏi một vài hạ tầng cơ sở phụ trợ. Trong một cốt lõi (nutshell), một Web Service đòi hỏi sau đây:

- Một wire protocol (nghĩa là HTTP GET / HTTP POST hoặc SOAP)
- Một dịch vụ mô tả (như vậy khách hàng biết Web Service có thể làm được gì).
- Một dịch vụ khám phá phát hiện (discovery service) như vậy khách hàng biết là Web Service nào đó hiện hữu.

Trong chương này bạn sẽ xem xét chi tiết của từng đòi hỏi kể trên. Tuy nhiên, trước khi đi sâu vào vấn đề, sau đây là một tổng quan ngắn gọn của mỗi công nghệ hỗ trợ.

## 9.2.1 Wire Protocol là gì?

Giống như với DataSet trên ADO.NET, thông tin được chuyển đi đi về về giữa phía tiêu thụ Web Service và phía dịch vụ Web Service nằm dưới dạng XML. Như đã nói, HTTP là nghi thức chuyển loại dữ liệu này. Đặc biệt là, bạn có thể dùng HTTP GET, HTTP POST, hoặc SOAP để di chuyển thông tin giữa Web Service và người tiêu thụ Web Service. Nhìn chung, người ta thích chọn SOAP vì các thông điệp SOAP có thể chứa mô tả XML đối với những kiểu dữ liệu phức tạp (các lớp custom, DataSet của ADO.NET, bản dãy các đối tượng, v.v..).

## 9.2.2 Web Service Description Services là gì?

Đối với một ứng dụng tiêu thụ Web Service, muốn sử dụng một Web Service nằm ở xa, nó phải hiểu thấu hoàn toàn những thành viên được trưng ra. Thí dụ, client phải biết là có một hàm hành sự mang tên Foo() mang 3 thông số kiểu dữ liệu {string, bool, int} và trả về một kiểu dữ liệu mang tên Bar trước khi triệu gọi hàm. Một lần nữa, XML lại nhập cuộc đem lại một cách thức chung (generic) để mô tả Web Service. Về mặt hình thức, XML schema được dùng để mô tả một Web Service nên mang tên là Web Service Description Language (WSDL).

## 9.2.3 Discovery Services là gì?

Trong chương trước, chúng tôi đã đề cập một cách ngắn ngủi tập tin \*.vsdisco (viết tắt DISCOevery of Web Service). Các tập tin XML này cho phép một đoạn mã client (hoặc một wizard thiết kế) khám phá một cách động các Web Service được trưng ra từ một địa chỉ URL nào đó. Cuối chương này, bạn sẽ làm quen với cú pháp của tập tin \*.vsdisco.

## 9.3 Tìm hiểu Namespace Web Service

Như bạn có thể tưởng tượng, mỗi một yêu cầu đều được hỗ trợ bởi những kiểu dữ liệu khác nhau của .NET, được chứa trong những namespace được liệt kê trong bảng 9-1:

**Bảng 9-1: Web Services Namespaces**

Namespaces	Mô tả
<b>System.Web.Services</b>	Namespace này chứa tập hợp tối thiểu và trọn vẹn những kiểu dữ liệu cần thiết để xây dựng một Web Service.
<b>System.Web.Services.Description</b>	Các kiểu dữ liệu trong namespace này cho phép bạn tương tác bằng lập trình với WSDL.
<b>System.Web.Services.Discovery</b>	Các kiểu dữ liệu trong namespace này (được dùng phối hợp với tập tin *.vsdisco) cho phép Web consumer bằng lập trình khám phá ra những Web Service được cài đặt trên một máy tính nào đó.
<b>System.Web.Services.Protocols</b>	Dữ liệu dạng thức XML được trao đổi giữa một Web consumer và một Web Service có thể được truyền đi sử dụng một trong 3 nghi thức (HTTP GET, HTTP POST hoặc SOAP). Namespace này định nghĩa một số kiểu dữ liệu tượng trưng cho những nghi thức này.

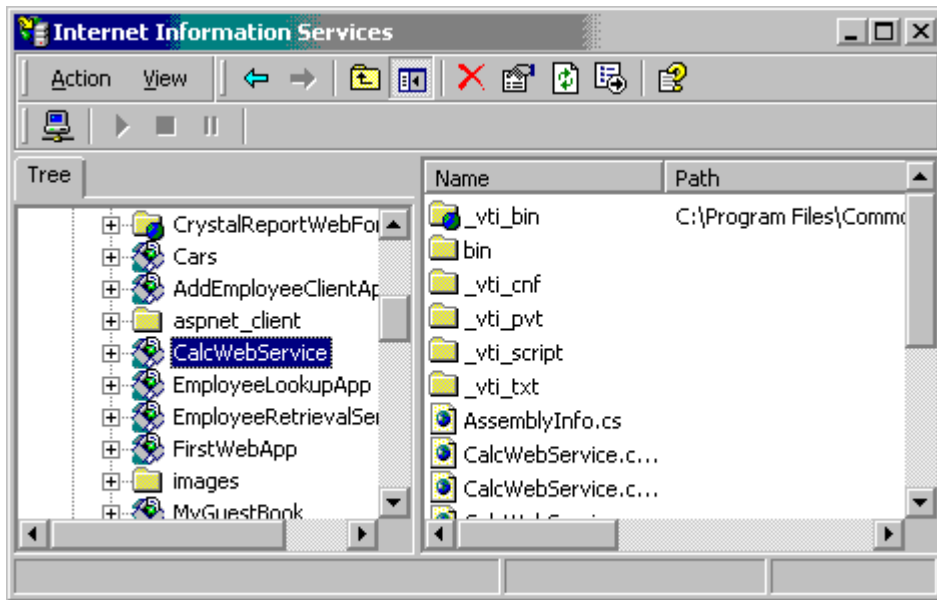
### 9.3.1 Thử xét namespace System.Web.Services

Mặc dù các namespace .NET Web Service cung cấp chức năng phong phú, đối với phần lớn các dự án, các kiểu dữ liệu mà bạn sẽ cần đến tương tác trực tiếp đều đã được định nghĩa trong namespace **System.Web.Services**. Như bạn có thể thấy từ bảng 9-2, số kiểu dữ liệu rất ư là ít.

**Bảng 9-2: Thành viên của namespace System.Web.Services.**

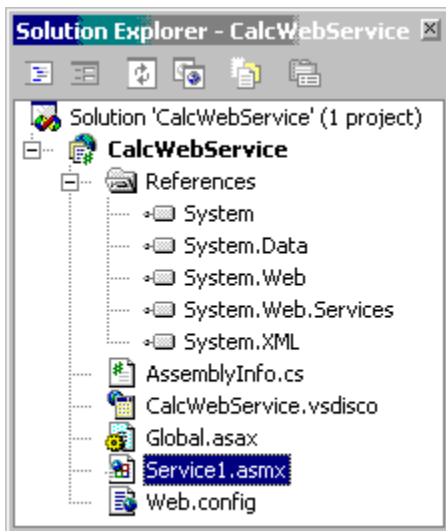
Các kiểu dữ liệu	Mô tả
<b>WebMethodAttribute</b>	Thêm attribute [WebMethod] vào một hàm hành sự trên Web Service làm cho hàm hành sự này có thể được triệu gọi từ một client nằm ở xa sử dụng HTTP.
<b>WebService</b>	Cho định nghĩa optional base class đối với Web Service.
<b>WebServiceAttribute</b>	Attribute [WebService] này có thể được dùng để thêm thông tin cho một Web Service, chẳng hạn một chuỗi mô tả chức năng của nó. Thuộc tính này không bắt buộc đối với một Web Service phải được công bố và thi hành.
<b>WebServiceBindingAttribute</b>	Thuộc tính này khai báo nghi thức gắn kết đối với một hàm hành sự Web Service nào đó phải thi công.

### 9.3.1.1 Xây dựng một Web Service đơn giản



Hình 9-01: Web Services được cài đặt bởi IIS

Trước khi nhảy bỏ vào chi tiết, ta thử xây dựng một thí dụ đơn giản. Bạn mở Visual Studio .NET, rồi tạo một C# Web service project mang tên **CalcWebService**. Bạn ra lệnh **File | New | Project**, rồi chọn **Visual C# project** (trên khung Project Types) và **ASP.NET Web Services** (trên khung Template), rồi đặt tên **CalcWebService** trên ô location. Bạn có lỗi tìm về: <http://localhost/CalcWebService>.



Hình 9-2: Các tập tin ban đầu

Giống như một ứng dụng ASP.NET, các dự án Web Service sẽ tự động tạo một thư mục ảo trên IIS (hình 9-1) và cho trữ các tập tin dự án dưới thư mục **C:\My Documents\Visual Studio Projects**.

Vì cấu hình của một dự án Web Service, nếu bạn muốn sử dụng một đoạn mã nguồn có thể nạp xuống khi làm việc với chương này, bạn bắt đầu tạo một workspace dự án mới, rồi đơn giản import lớp được định sẵn trước. Trong bất cứ trường hợp nào, khi bạn khảo sát Solution Explorer (hình 9-2), bạn sẽ thấy thoải mái với



những tập tin thuộc dự án như đã được trình bày trong chương trước.

Các tập tin **Global.asax** và **Web.config** được dùng cùng mục đích như với một ứng dụng ASP.NET. Như bạn đã nhớ trong chương trước, tập tin **Global.asax** cho phép bạn phản ứng trước những tình huống mang tính toàn cục. Còn tập tin **Web.config** cho phép bạn cấu hình hóa bằng cách khai báo Web Service mới của bạn (sử dụng lần nữa ký hiệu XML). Các mục mà chúng tôi quan tâm tới lúc này là các tập tin \*.asmx, \*.asmx.cs và \*.vsdisco, như được mô tả trên bảng 9-3.

**Bảng 9-3: Các tập tin cốt lõi của một Visual Studio .NET Web Service**

Các tập tin	Mô tả
*.asmx *.asmx.cs	Các tập tin này định nghĩa các hàm hành sự của Web Service. Giống như với một tập tin *.aspx, mỗi tập tin *.asmx sẽ có một tập tin *.cs tương ứng để trữ phần code behind.
*.vsdisco	Tập tin này chứa phần mô tả của Web Service (tại một URL nào đó) dưới dạng XML.

### 9.3.1.1.1 Tập tin Codebehind (\*.asmx.cs)

Một tập tin \*.asmx tượng trưng cho một Web Service nào đó trong workspace dự án của bạn. Muốn xem đoạn mã nằm ẩn sau template thiết kế, bạn chọn mục View Code để nhìn xem phần định nghĩa lớp \*.asmx.cs tương ứng.

```
public class Service1: System.Web.Services.WebService
{
    public Service1() { InitializeComponent(); }

    private void InitializeComponent() {}

    protected override void Dispose( bool disposing ) {}
};
```

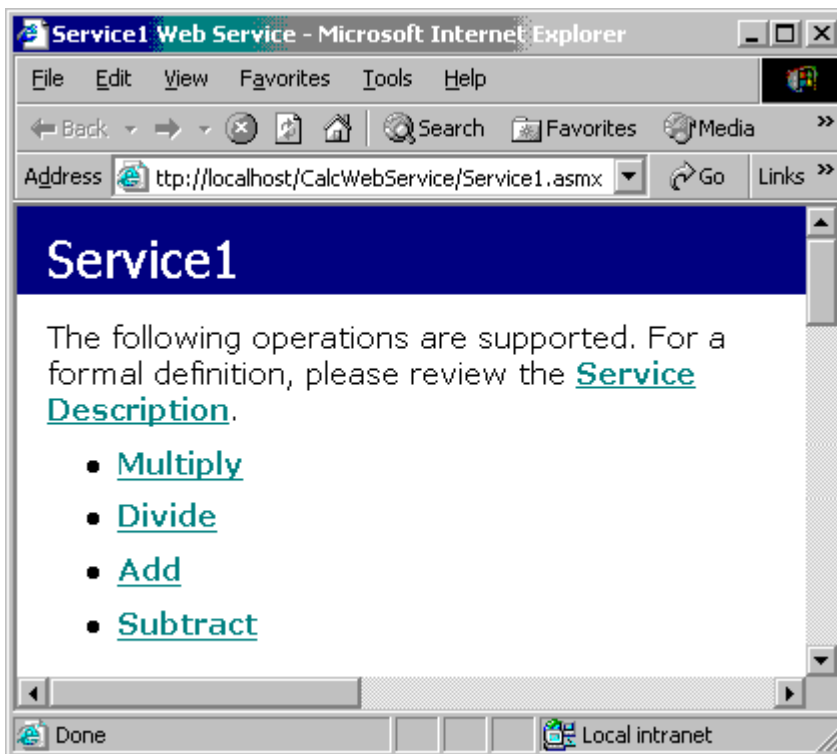
Như bạn có thể thấy, điểm thực sự đáng quan tâm là việc bạn dẫn xuất từ một lớp cơ bản mới: **WebService**. Trong chốc lát bạn sẽ xét đến các thành viên được định nghĩa bởi lớp này. Trong lúc này, bạn chỉ cần biết Web Service có một *mục chọn* (option) dẫn xuất từ kiểu dữ liệu lớp cơ bản. Thật thế, nếu bạn comment out hàm hành sự **Dispose()** và dẫn xuất trực tiếp từ **Sytem.Object**, thì Web Service vẫn hoạt động ngon lành, như sau:

```
// Ta vẫn còn là một Web service
public class Service1
{
    public Service1() { InitializeComponent(); }
    private void InitializeComponent() {}
    // protected override void Dispose( bool disposing ) {}
};
```

### 9.3.1.1.2 Thêm một vài chức năng đơn giản

Đối với Web Service ban đầu này, để cho đơn giản ta thêm 4 hàm ngắn gọn tượng trưng cho cộng trừ nhân chia hai số nguyên. Như bạn có thể chờ đợi, các hàm hành sự bạn mong muốn có sẵn thông qua yêu cầu theo HTTP phải được khai báo là public, và mỗi hàm hành sự phải hỗ trợ attribute **[WebMethod]**. Do đó, bạn phải nhật tu lớp ban đầu của bạn như sau:

```
public class Service1: System.Web.Services.WebService
{
    public Service1() { InitializeComponent(); }
    private void InitializeComponent() {}
    protected override void Dispose( bool disposing ) {}
    [WebMethod]
    public int Add(int x, int y) {return x + y;}
    [WebMethod]
    public int Subtract(int x, int y) {return x - y;}
    [WebMethod]
    public int Multiply(int x, int y) {return x * y;}
    [WebMethod]
    public int Divide(int x, int y) {return x / y;}
};
```

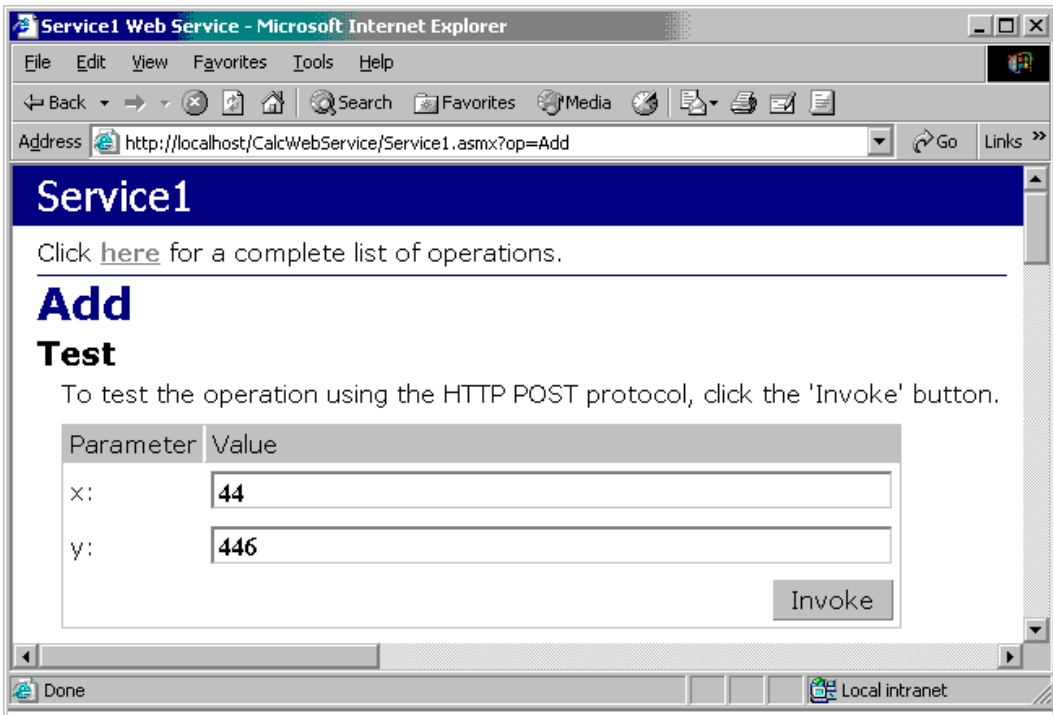


Hình 9-3: IE cho phép Bạn trải nghiệm nhanh Web Services

### 9.3.1.1.3 Trắc nghiệm Web service

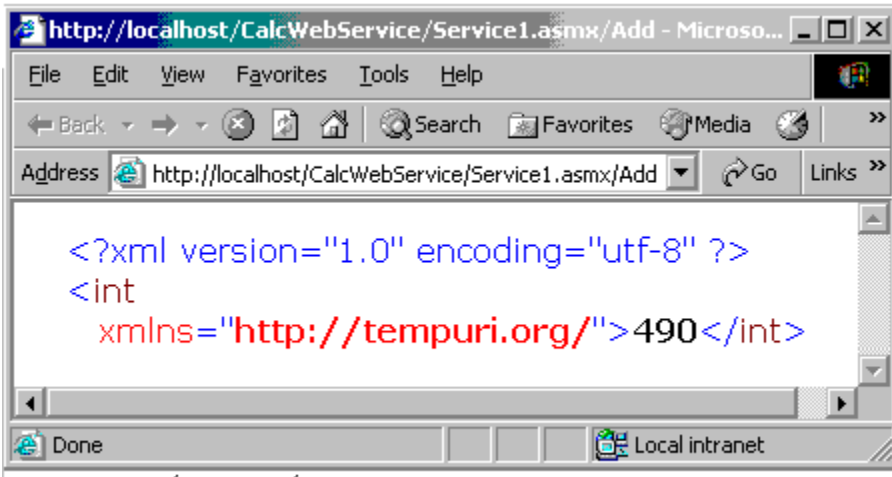
Một khi bạn biên dịch xong Web Service, bạn có thể cho thi hành ngay sử dụng Visual Studio .NET IDE, nghĩa là chỉ cần **Build** rồi **Debug | Start**. Hình 9-3 cho thấy kết quả trắc nghiệm. Theo mặc nhiên, browser hiện dịch trên máy của bạn hoạt động như là một khách hàng tạm thời thay thế, cho thấy một HTML view của các hàm hành sự với thuộc tính [**WebMethod**]. Trên hình 9-3 trang trước, bạn thấy có 4 hàng có mang dấu chấm đen: Multiply, Divide, Add, và Subtract. Nếu con nháy chuột rề rề lên một trong 4 hàng này, nó sẽ ngời sáng đổi màu. Ta gọi mỗi hàng có thể ngời sáng này là một link.

Ngoài việc liệt kê mỗi hàm hành sự được định nghĩa trong một Web Service nào đó, bạn có thể triệu gọi trực tiếp mỗi hàm hành sự từ trong lòng browser. Thí dụ, bạn click **Add** link rồi khổ vào vài trị số (hình 9-4). Như bạn có thể thấy, HTML cho hiển thị các ô text box cho phép người sử dụng khổ vào dữ liệu.



Hình 9-4: IE cho phép Bạn triệu gọi một hàm Web với thông số

Khi bạn triệu gọi hàm kết quả (490) sẽ được trả về thông qua một thuộc tính XML (hình 9-5).



Hình 9-5: Kết quả cuối cùng

Một điểm khá lý thú là cách thông tin được gửi cho Web Service thế nào. Nếu bạn cho kiểm tra query string được kết sinh bạn sẽ thấy như sau:

```
http://localhost/CalcWebServie/Service1.asmx/Add?x=44&y=446
```

Bạn để ý là URL được hình thành bởi tên hàm hành sự được triệu gọi (Add) theo sau là tên các thông số (và trị).

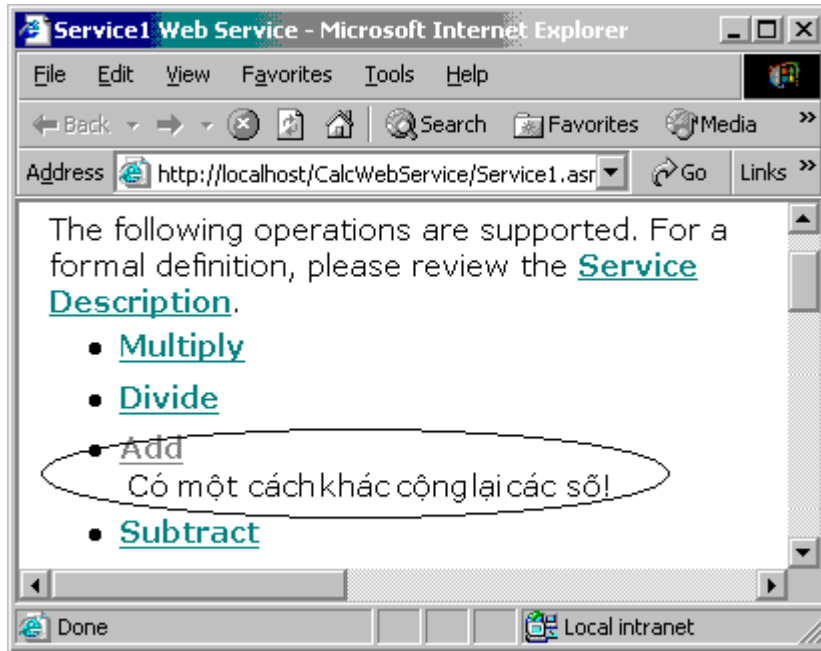
Như bạn có thể thấy, tương đối dễ xây dựng và trắc nghiệm một Web Service. Về sau vào cuối chương này, bạn sẽ học cách tạo một Web Service phức tạp hơn. Trong khi chờ đợi, ta tiếp tục khảo sát các chi tiết nằm sau kiến trúc Web Service.

### 9.3.1.2 Lớp *WebMethodAttribute*

Attribute [WebMethod] phải được áp dụng đối với mỗi hàm hành sự mà bạn muốn cho trung ra cho thế giới bên ngoài thông qua HTTP. Giống như phần lớn các attribute, kiểu dữ liệu **WebMethod** có thể có một số thông số hàm constructor tùy chọn. Thí dụ, muốn mô tả chức năng của một hàm hành sự Web, cú pháp sau đây có thể được dùng:

```
[WebMethod(Description = "Có một cách khác cộng lại các số!")]  
public int Add(int x, int y) {return x+y;}
```

Trong chừng mực nào đó, cho đặt để khía cạnh **Description** của attribute [WebMethod] cũng tương tự như attribute [helpstring] của IDL. Nếu bạn cho Build rồi Run thì bạn có hình 9-6 như sau, dòng chữ: "Có một cách khác cộng lại các số!" dưới lệnh Add.



**Hình 9-6: Kết quả đặt để thuộc tính WebMethod.Description**

Sau hậu trường, khế ước WSDL sẽ được nhật tu với một thuộc tính <documentation> mới như sau:

```
<operation name="Add">
  <input message="s0:AddSoapIn" />
  <output message="s0:AddSoapOut" />
  <documentation>Có một cách khác cộng lại các số!</documentation>
</operation>
```

Ngoài khía cạnh **Description**, bạn có thể cấu hình hóa thuộc tính WebService sử dụng bất cứ thuộc tính cốt lõi được mô tả trong bảng 9-4:

**Bảng 9-4: WebServiceAttribute**

Các thuộc tính	Mô tả
<b>Description</b>	Dùng thêm một đoạn văn bản thân thiện mô tả hàm hành sự Web.
<b>EnableSession</b>	Theo mặc nhiên, thuộc tính này được cho về true, yêu cầu hàm hành sự này duy trì trạng thái châu giao dịch (session state). Bạn có thể cho đặt về false nếu bạn muốn vô hiệu lực khả năng này.
<b>MessageName</b>	Thuộc tính này có thể đem dùng để cấu hình một hàm hành sự Web được tượng trưng thể nào trên WSDL để tránh đụng độ về tên.
<b>TransactionOption</b>	Các hàm hành sự Web có thể hoạt động như là gốc của một giao dịch COM+. Thuộc tính này có thể được gán cho bất cứ trị nào từ enumeration System.EnterpriseServices.TransactionOption.

Một item đặc biệt đáng chú ý là attribute **MessageName**. Để minh họa, giả sử Web Calculator của bạn giờ đây định nghĩa một hàm hành sự bổ sung có thể cộng hai số float như sau:

```
[WebMethod(Description="Cộng hai số nguyên")]
public int Add(int x, int y) {return x + y;}

[WebMethod(Description="Cộng hai số thực")]
public float Add(float x, float y) {return x + y;}
```

Nếu bạn biên dịch thì việc biên dịch thành công, không kết sinh sai lầm, nhưng khi cho chạy trên Web thì bạn sẽ nhận thông điệp sai lầm như sau cho biết cả hai hàm đều dùng một tên Add:

**Both Single Add(Single, Single) and Int32 Add(Int32, Int32) use the message name 'Add'. Use the MessageName property of the WebMethod custom attribute to specify unique message names for the methods.**

Một đòi hỏi của WSDL là mỗi attribute **<soap:operation soapAction>** (dùng định nghĩa tên của một hàm hành sự Web nào đó) phải là duy nhất về mặt tên. Tuy nhiên, hành xử mặc nhiên của WSDL generator là kết sinh tên **<soap:operation soapAction>** đúng y chang như được xuất hiện trên định nghĩa mã nguồn. Do đó bạn có hai hàm Web cùng mang tên Add. Để giải quyết vấn đề đụng độ tên, bạn có thể hoặc đặt tên khác hoặc sử dụng attribute **MessageName** để thiết lập tính duy nhất về tên:

```
[WebMethod(Description="Cộng hai số nguyên")]
public int Add(int x, int y) {return x + y;}

[WebMethod(Description="Cộng hai số thực", MessageName = AddFloats)]
public float Add(float x, float y) {return x + y;}
```

Với cách làm này, bạn có thể thấy là mỗi mô tả WSDL giờ đây là duy nhất:

```
<operation name="Add">
  <soap:operation soapAction=http://tempuri.org/AddFloats
    style="document"/>
  <input name="AddFloats">
    <soap:body use="literal"/>
  </input>
  <output name="AddFloats">
    <soap:body use="literal"/>
  </output>
</operation>

<operation name="Add">
  <soap:operation soapAction=http://tempuri.org/Add
    style="document"/>
  <soap:body use="literal"/>
</operation>
```

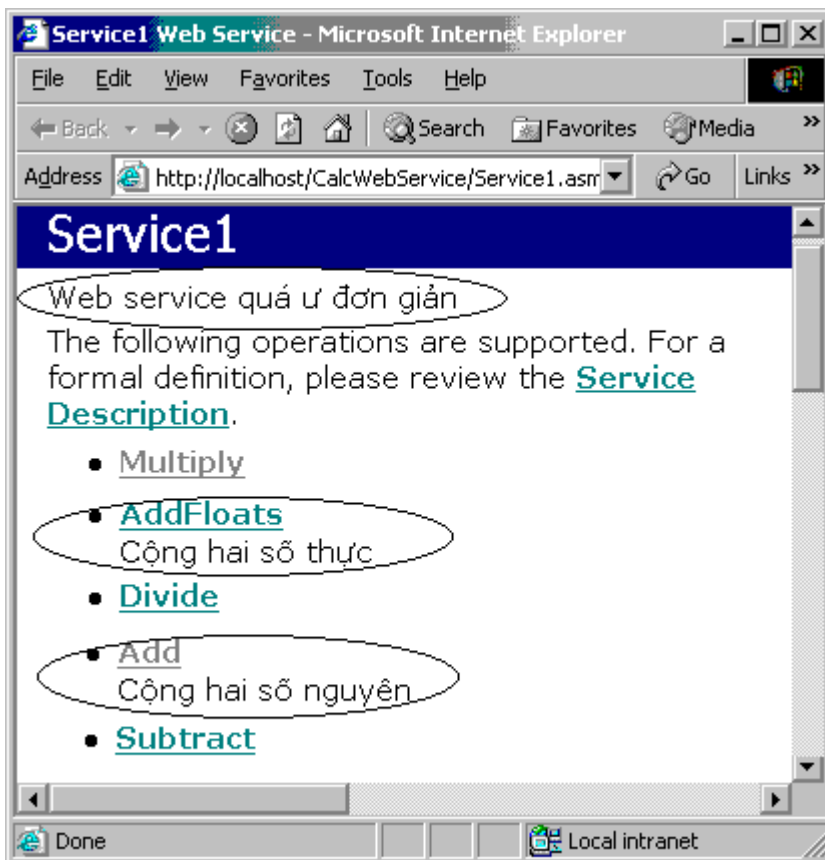
```
</input>
  <soap:body use="literal"/>
</output>
</operation>
```

Nếu bạn cho **Build** và chạy lại, hình 9-7 đi sau cho thấy kết quả của hai hàm **Add**.

WebServiceAttribute còn cung cấp cho bạn một thuộc tính Description cho phép bạn sru liệu chức năng tổng quát của bản thân Web Service, như sau:

```
[WebService(Description="Web Service rất ư đơn giản ")]
public class Service1: System.Web.Services.WebService
{
...
}
```

Nếu cho Build và Debug, bạn sẽ thấy hình 9-7:



Hình 9-7: WebServiceAttribute mô tả bản chất service

### 9.3.1.3 Lớp cơ bản *System.Web.Services.WebService*

Như đã nói trước đây, .NET Web Service được tự do dẫn xuất trực tiếp từ **System.Object**. Tuy nhiên, theo mặc nhiên Web Service được triển khai sử dụng Visual Studio .NET sẽ tự động dẫn xuất từ lớp cơ bản **WebService**. Chức năng được cung cấp bởi lớp này trang bị cho Web Service của bạn tương tác với cùng kiểu dữ liệu được sử dụng bởi mô hình đối tượng ASP.NET (bảng 9-5).

**Bảng 9-5: Các thuộc tính cốt lõi của lớp *WebService* cơ sở**

Các thuộc tính	Mô tả
<b>Application</b>	Thuộc tính này đi lấy một qui chiếu về đối tượng ứng dụng đối với HTTP request hiện hành.
<b>Context</b>	Thuộc tính này đi lấy một đối tượng ASP.NET Context đối với yêu cầu hiện hành, phạm trù này gói ghém tất cả các phạm trù đặc trưng HTTP được dùng bởi server HTTP để xử lý các yêu cầu Web.
<b>Server</b>	Thuộc tính này đi lấy một qui chiếu về <b>HttpServerUtility</b> đối với yêu cầu hiện hành.
<b>Session</b>	Thuộc tính này đi lấy một qui chiếu về thể hiện <b>SessionState</b> . <b>HttpSessionState</b> đối với yêu cầu hiện hành.
<b>User</b>	Thuộc tính này đi lấy một đối tượng ASP.NET Server User có thể được dùng để chứng thực (authenticate) một người sử dụng nào đó.

Như bạn còn nhớ lại trong chương trước, các thuộc tính **Application** và **Session** cho phép bạn duy trì dữ liệu trạng thái trong suốt thời gian thi hành ứng dụng ASP.NET. Web Service cũng cung cấp chức năng y chang. Thí dụ, giả sử **CalcWebService** duy trì một biến cấp ứng dụng (nghĩa là có sẵn cho mỗi session), cầm giữ trị của PI, như sau:

```
public class Service1: System.Web.Services.WebService
{
    public Service1()
    {
        InitializeComponent();
        Application["SimplePI"] = 3.14F;
    }

    private void InitializeComponent() {}

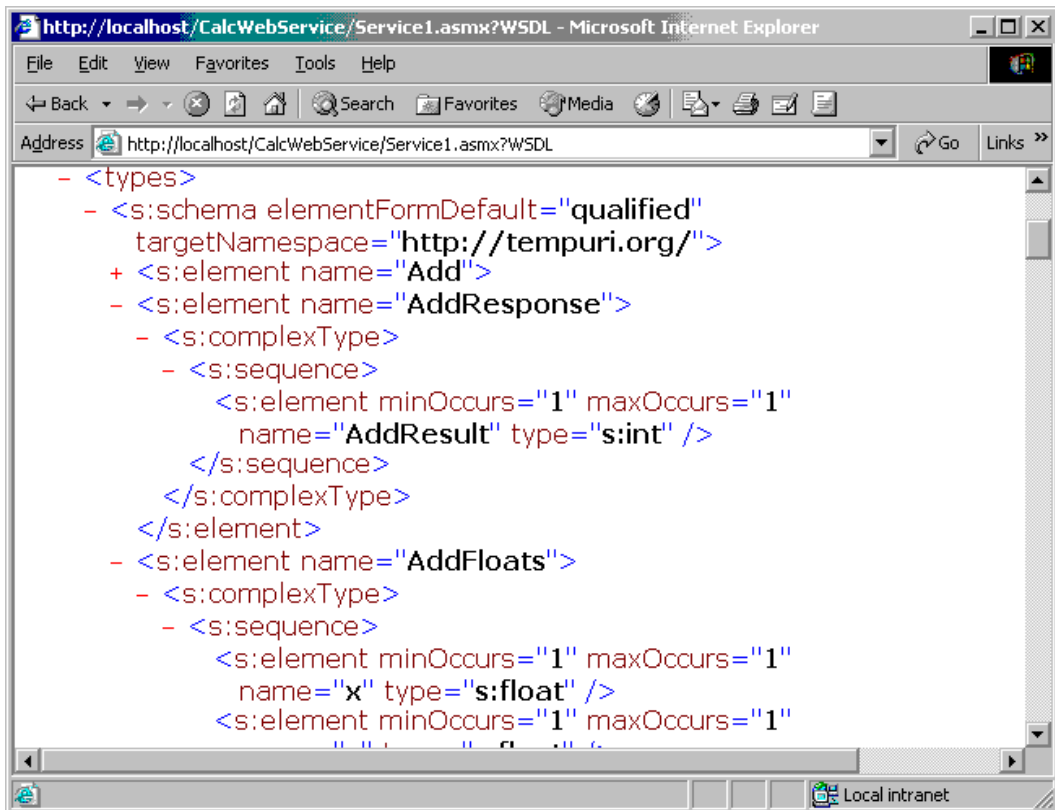
    protected override void Dispose( bool disposing ) {}

    [WebMethod]
    public float GetSimplePI()
    {
        return (float)Application["SimplePI"];
    }
    . . .
}
```



## 9.3.2 Thử tìm hiểu namespace System.Web.Services.Description

Bây giờ bạn đã biết một Web Service đơn giản hoạt động thế nào, ta thử tiếp tục xem các hàm hành sự Web được mô tả sau hậu trường. Lập trình viên COM thường hiểu IDL là một metalanguage dùng định nghĩa mỗi khía cạnh của một COM item. Còn lập trình viên .NET cũng hiểu rằng trình biên dịch tạo ra managed code cũng phát ra metadata đầy đủ trọn vẹn mô tả trọn vẹn các kiểu dữ liệu trong assembly. Khi một hình ảnh nhị phân (COM hoặc .NET) được mô tả trong những từ ngữ độc lập so với ngôn ngữ, coi như bạn thiết lập một khế ước là khách hàng có thể đọc để hiểu các qui ước triệu gọi hàm, tên kiểu dữ liệu, kiểu dữ liệu cơ bản, các giao diện được hỗ trợ v.v..



Hình 9-8: Raw WSDL

Trong tinh thần của IDL và metadata .NET, Web Service cũng mô tả sử dụng metalanguage mang tên WSDL (Web Service Description Language). WSDL là một tập hợp con của XML được mô tả trọn vẹn cách các khách hàng nằm ngoài có thể tương tác

với Web Service trên một máy tính nào đó, các hàm hành sự nó chịu hỗ trợ và cú pháp của các nghi thức truyền tin khác nhau (GET, POST hoặc SOAP).

Khi bạn trải nghiệm Web Service của bạn từ browser được chọn ra, bạn sẽ thấy một link mang tên **Service Description** trên các hình 9-3, 9-6 và 9-7.

Khi bạn click lên link này, một cửa sổ riêng biệt sẽ hiện lên mô tả khế ước được định nghĩa bởi Web Service hiện hành (hình 9-8, trang trước).

WSDL là một khế ước được mở và được đóng lại sử dụng các tag <definitions>. Sau tag <definitions>, mở đầu, là một loạt những nodes định nghĩa các wire protocols khác nhau, như sau:

```
<?xml version="1.0" encoding="utf-8" ?>
<definitions xmlns:http="http://schemas.xmlsoap.org/wsdl/http/"
  xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
  xmlns:s="http://www.w3.org/2001/XMLSchema"
  xmlns:s0="http://tempuri.org/"
  xmlns:soapenc="http://schemas.xmlsoap.org/soap/encoding/"
  xmlns:tm="http://microsoft.com/wsdl/mime/textMatching/"
  xmlns:mime="http://schemas.xmlsoap.org/wsdl/mime/"
  targetNamespace="http://tempuri.org/"
  xmlns="http://schemas.xmlsoap.org/wsdl/">
...

```

Tiếp theo, bạn sẽ thấy WSDL definition đối với mỗi hàm hành sự Web được định nghĩa theo GET, POST và SOAP wire protocols (hình 9-9):

Như bạn có thể thấy, mỗi hàm hành sự Web sẽ có một In và Out. Do đó, hàm hành sự Web Subtract() sẽ có 6 tag <message name>: một cặp In/Out cho HTTP POST, một cặp cho HTTP GET và một cặp khác cho SOAP. Thí dụ, với SOAP bạn có:

```
<message name="SubtractSoapIn">
  <part name="parameters" element="s0:Subtract" />
</message>

<message name="SubtractSoapOut">
  <part name="parameters" element="s0:SubtractResponse" />
</message>

```

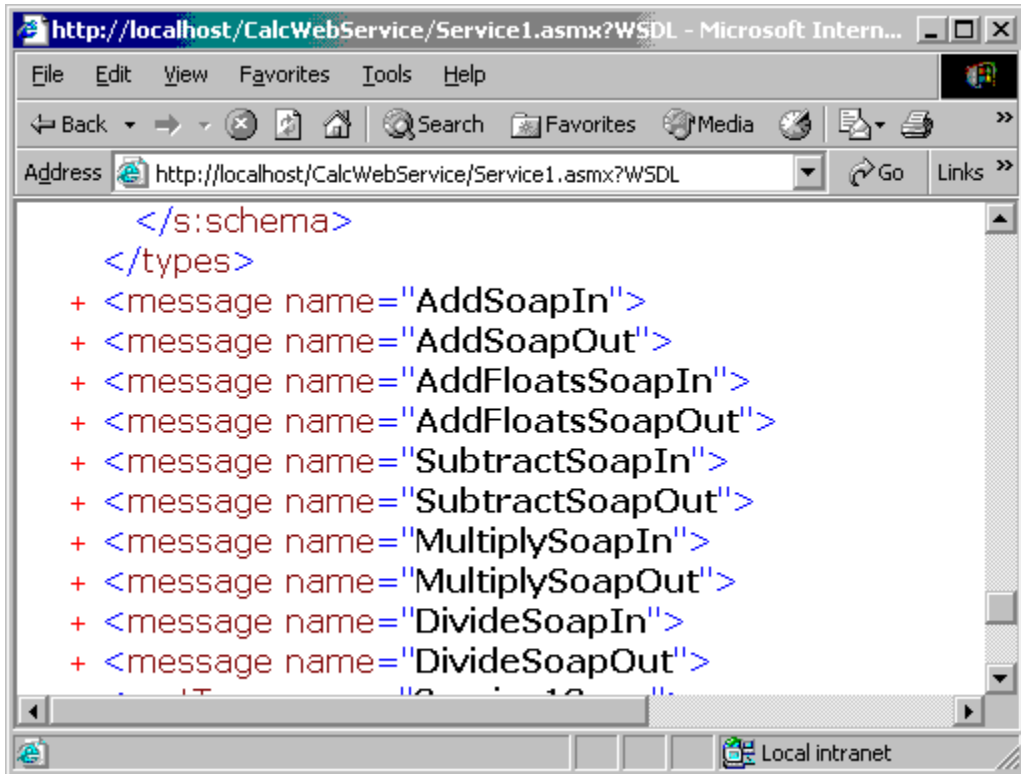
Còn với HTTP POST protocol bạn lại có như sau:

```
<message name="SubtractHttpPostIn">
  <part name="x" type="s:string" />
  <part name="y" type="s:string" />
</message>

<message name="SubtractHttpPostOut">

```

```
<part name="Body" element="s0:int" />
</message>
```



Hình 9-9: Mỗi hàm Web đều có những cặp GET, POST và SOAP

Một lần nữa, giống như những metalanguage khác, WSDL chỉ trực tiếp quan tâm nếu bạn đang xây dựng một cái gì đó như custom parser hoặc type viewer (trình nhìn xem các kiểu dữ liệu). Nếu đúng là nghề của bạn thì bạn nên dành thời gian khảo sát namespace **System.Web.Services.Description**. Ở đây bạn thấy vô số kiểu dữ liệu cho phép bạn tung hoành về mặt lập trình thao tác WSDL. Còn nếu bạn không quan tâm, thì bạn nhớ cho WSDL mô tả trọn vẹn các qui ước triệu gọi hàm (và nghi thức truyền dữ liệu) cho phép khách hàng nằm bên ngoài triệu gọi hàm Web được định nghĩa bởi một Web Service.

### 9.3.3 Thử xét namespace **System.Web.Services.Protocols**

Như bạn đã biết, mục đích của Web Service là trả về dữ liệu ở dạng XML về cho phía tiêu thụ, sử dụng nghi thức HTTP. Đặc biệt, một Web server cho dữ liệu này nằm

trong một cái thân của một thông điệp HTTP rồi chuyển đi cho phía tiêu thụ sử dụng một trong 3 kỹ thuật đặc biệt (bảng 9-6):

**Bảng 9-6: Web Service Wire Protocols**

Nghi thức chuyển tin	Mô tả
<b>HTTP GET</b>	Các GET submission cho ghi nối đuôi các thông số vào query string của URL hiện hành.
<b>HTTP POST</b>	POST transmission cho đặt lọt thỏm các data point vào header của thông điệp HTTP thay vì cho ghi nối đuôi vào query string.
<b>SOAP</b>	SOAP là một wire protocol khai báo cho biết làm thế nào submit dữ liệu xuyên mạng sử dụng XML.

Mặc dù mỗi cách tiếp cận đều đi đến cùng kết quả (triệu gọi hàm nằm ở xa sử dụng HTTP), việc bạn chọn loại nghi thức truyền tin nào sẽ xác định các loại thông số (và kiểu dữ liệu trả về) có thể được chuyển đi giữa các bên liên quan. Nghi thức SOAP sẽ đem lại cho bạn dạng uyển chuyển nhất. Tuy nhiên, để cho đầy đủ, chúng tôi bắt đầu xem xét việc sử dụng mã hóa theo GET và POST chuẩn.

### 9.3.3.1 Chuyển tin sử dụng nghi thức HTTP GET và HTTP POST

Như đã giải thích trong chương trước, nghi thức HTTP GET được chuyển cho nơi tiếp nhận (recipient) bằng cách ghi nối đuôi những cặp name/value vào cuối của URL tiếp nhận. Bạn còn nhớ dấu chấm hỏi (?) cho ngăn cách trang khỏi tập hợp các thông số mang tên. Còn nghi thức HTTP POST thì tượng trưng cho dữ liệu đi vào như là những cặp name/value được đặt vào thân của thông điệp HTTP. Khi bạn sử dụng việc chuyển tin theo nghi thức HTTP GET hoặc POST, kết quả cuối cùng giống như nhau. Trị được trả về sẽ ở dưới dạng XML đơn giản theo dạng <type>VALUE<type>.

Mặc dù các động từ GET và POST có thể là những cấu trúc quen thuộc, có thể bạn cảm nhận là kiểu chuyển tin này không đủ phong phú để tượng trưng cho những item phức tạp như các structures hoặc thể hiện đối tượng. Khi bạn sử dụng GET và POST, bạn chỉ có thể tương tác với các hàm hành sự Web sử dụng các kiểu dữ liệu sử dụng (bảng 9-7).

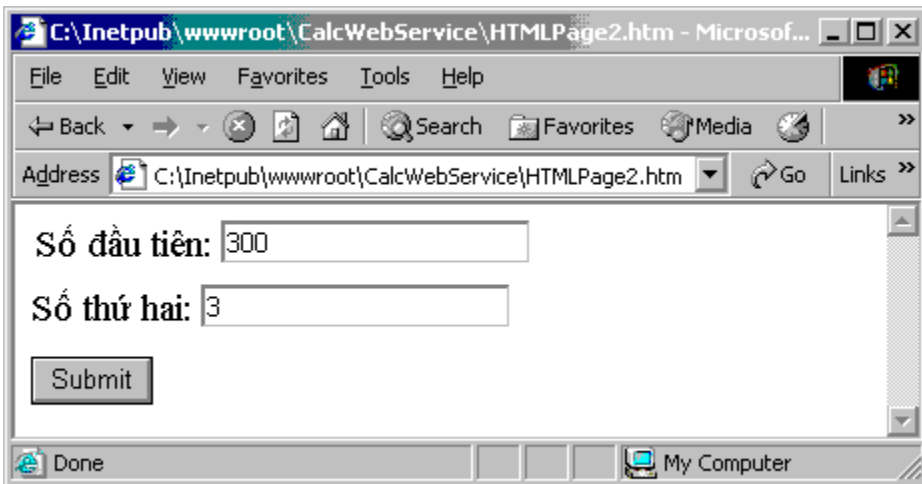
**Bảng 9-7: Các kiểu dữ liệu được hỗ trợ bởi GET và POST**

Các kiểu dữ liệu	Mô tả
<b>Enumerations</b>	GET và POST chịu hỗ trợ chuyển tin của các kiểu dữ liệu System.Enum. Các kiểu dữ liệu này được tượng trưng bởi những chuỗi hằng static.
<b>Bản dãy đơn giản</b>	Bạn có thể xây dựng những bản dãy của bất cứ kiểu dữ liệu bẩm sinh.
<b>Strings</b>	GET và POST sẽ chuyển đi bất cứ dữ liệu kiểu số dưới dạng string. String thực sự ám chỉ chuỗi biểu diễn các kiểu dữ liệu bẩm sinh của

CLR chẳng hạn Int16, Int32, Int64, Boolean, Single, Double, Decimal, DateTome, v.v..
--

Muốn xây dựng một biểu mẫu HTML lo trình duyệt (submit) dữ liệu sử dụng POST hoặc GET, bạn khai báo tập tin \*.asmx như là nơi chứa chấp dữ liệu của biểu mẫu. Giả sử tập tin \*.htm sau đây tạo một UI cho phép người sử dụng khở vào số liệu gửi cho hàm hành sự Subtract() của ValcWebService, sử dụng nghi thức GET:

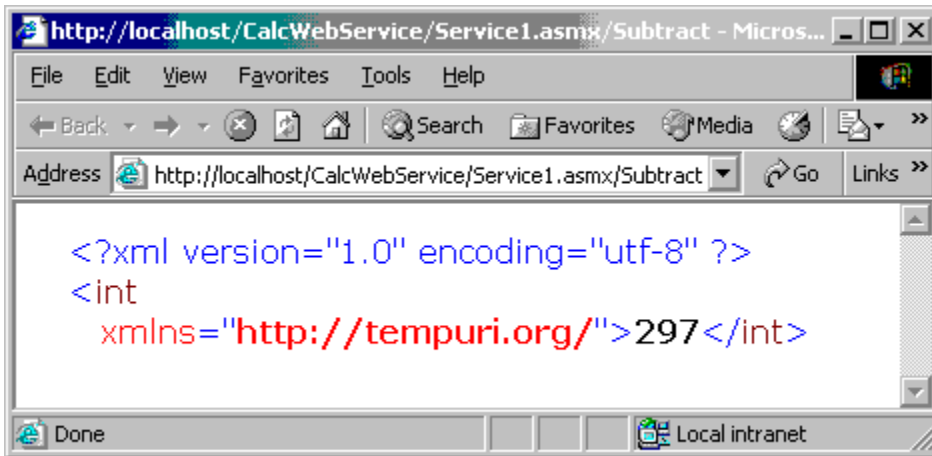
```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0 Transitional//EN">
<html>
<head>
<title></title>
<meta name="GENERATOR" content="Microsoft Visual Studio.NET 7.0">
<meta name="vs_targetSchema"
      content="http://schemas.microsoft.com/intellisense/ie5">
</head>
<body>
  <form method = 'GET' action
    ='http://localhost/CalcWebService/CalcService.asmx/Subtract'>
    <p>Số đầu tiên:
      <input id=Text1 name=x type=text></p>
    <p>Số thứ hai:
      <input id="Text2" name=y type=text></p>
    <p><input id=Submit1 type=submit value=Submit</p>
  </body>
</html>
```



**Hình 9-10: Làm phép trừ sử dụng HTTP GET**

Một vài điểm đáng lưu ý. Trước tiên, action attribute chỉ về không những tập tin \*.asmx, mà còn về tên hàm hành sự cần triệu gọi. Tiếp theo, bạn để ý là name attribute được dùng nhận diện tên của mỗi thông số. Chắc bạn còn nhớ, hàm Subtract() có hai số nguyên mang tên x và y.

Hình 9-10 cho thấy kết quả khỏ vào hai số 300 và 3 như là dữ liệu nhập. Hình 9-11 là kết quả khi bạn ấn nút <Submit>.



Hình 9-11: Kết quả trừ các số

### 9.3.3.2 Chuyển tin sử dụng nghi thức SOAP

Một phương án di chuyển dữ liệu hấp dẫn hơn giữa phía tiêu thụ và Web Service là sử dụng SOAP, cho phép tương trưng nhiều kiểu dữ liệu phức tạp (dựa theo ký hiệu XML) như theo bảng 9-8:

**Bảng 9-8: Các kiểu dữ liệu SOAP**

Các kiểu dữ liệu	Mô tả
<b>ADO.NET DataSet</b>	Mặc dù DataSet là một lớp khác, nhưng điểm cần nhấn mạnh là kiểu dữ liệu này được hỗ trợ.
<b>Complex Array</b>	Bạn có thể xây dựng những bản dãy các lớp, structures và XML nodes.
<b>Custom Types</b>	Sử dụng SOAP bạn có thể xây dựng những hàm hành sự Web cho phép trưng ra những kiểu dữ liệu “cây nhà lá vườn”.
<b>XML Nodes</b>	Các hàm hành sự Web có thể cho trưng ra những XML nodes được di chuyển như là XML.

Mặc dù việc khảo sát một cách cặn kẽ SOAP là ngoài tầm tay của chương này, nhưng bạn hiểu cho là SOAP được thiết kế làm thế nào càng đơn giản càng tốt. Do đó, bản thân SOAP không định nghĩa một nghi thức nào đặc biệt nên có thể đem sử dụng với bất cứ nghi thức hiện hữu trên Internet (HTTP, SMTP, v.v..).

Về mặt cốt lõi, đặc tả SOAP chứa hai khía cạnh. Thứ nhất, là phần vỏ bọc ở ngoài. Thứ hai, ta có những qui tắc được sử dụng để mô tả thông tin của thông điệp này.

Bạn nhớ lại khi bạn sử dụng SOAP để triệu gọi hàm hành sự Add(), phần định nghĩa của SOAP như sau:

```
<message name="AddFloatsSoapIn">
  <part name="parameters" element="s0:AddFloats" />
</message>

<message name="AddFloatsSoapOut">
  <part name="parameters" element="s0:AddFloatsResponse" />
</message>
```

Bạn cho mở cửa sổ WSDL đối với CalcWebService. Vào cuối trang, bạn sẽ thấy 3 XML node mô tả GET, POST và SOAP binding. Nếu bạn bung SOAP binding đối với Web Service bạn sẽ thấy mô tả sau đây đối với hàm hành sự Add() (bạn ghi nhận tag in và out):

```
<operation name="Add">
<soap:operation soapAction=http://tempuri.org/AddFloats
style="document" />
  <input name="AddFloats">
    <soap:body use="literal" />
  </input>
  <output name="AddFloats">
    <soap:body use="literal" />
  </output>
</operation>
```

Tới đây coi như bạn đã thoải mái với .NET Web Service. Bước kế tiếp là tìm hiểu làm thế nào tạo ứng dụng client có thể tiêu thụ các Web Service này.

## 9.4 WSDL trong đoạn mã C# - Kết sinh một proxy

Như bạn đã thấy, WSDL được dùng mô tả các hàm hành sự Web theo cú pháp XML. Tuy nhiên, không nên tạo những khách hàng *yêu cầu bằng tay* một định nghĩa WSDL và phân tích ngữ nghĩa cũng *bằng tay* mỗi mắt gút (node) để thiết lập một kết nối với Web service nằm ở xa. Cách tiếp cận tốt nhất là làm thế nào có thể kết sinh một proxy đối với một hàm hành sự Web.

Proxy có thể đơn giản được định nghĩa như là một lớp hoạt động giống như một thực thể nằm ở xa mà nó tượng trưng. Nếu bạn đã kinh qua COM thì việc này không có chi là lạ lắm. Khi bạn gởi một tập tin IDL cho trình biên dịch MIDL, một trong những tập tin được kết sinh (\*\_p.c) chứa C-based stub code và proxy code mà ta có thể biên dịch thành một DLL nhị phân. COM client sẽ triệu gọi hàm trên lớp proxy (sử dụng ORPC

protocol). Proxy lo gói ghém các thông số đi vào rồi chuyển chúng đi về stub tiếp nhận. Đến phiên stub cho bạn gói yêu cầu ra trao yêu cầu cho đối tượng COM thực sự.

Cách hành xử cũng tương tự khi một ứng dụng tiêu thụ sử dụng một Web Service nằm ở xa. Khác biệt chủ yếu là việc truy cập một Web Service *không* tùy thuộc vào dạng thức nhị phân phía “chủ nhân”, hoặc một sản phẩm đặc biệt hoặc một ngôn ngữ lập trình nào đó. Tất cả các chi đòi hỏi hiểu thấy HTTP và XML.

Việc kết sinh một Web Service proxy khá đơn giản. Trước tiên, bạn có thể dùng một trình tiện ích mang tên **wsdl.exe** chạy trên command line. Một phương án thay thế khác là Visual Studio .NET IDE cho phép bạn qui chiếu về một Web Service bằng cách sử dụng một wizard thân thiện. Ta thử xem mỗi cách tiếp cận này.

## 9.4.1 Xây dựng một proxy sử dụng wsdl.exe

Trình tiện ích wsdl.exe, chạy trên command line, sẽ kết sinh một tập tin tượng trưng cho proxy đối với Web Service nằm ở xa. Tối thiểu bạn phải khai báo tên tập tin proxy cần được kết sinh và URL theo đây WSDL có thể nhận được như sau:

```
wsdl.exe /out:c:\calcproxy.cs
http://localhost/calculatwebservice/service1.asmx?WSDL
```

Trình tiện ích wsdl.exe sẽ kết sinh C# code theo mặc nhiên. Nếu bạn chọn một proxy viết theo VB.NET hoặc JScript.NET, bạn có thể dùng flag tùy chọn /l: (Language), bảng 9-9 liệt kê một vài flag tùy chọn đối với trình tiện ích wsdl.exe.

**Bảng 9-9: Các flag khác nhau đối với trình tiện ích wsdl.exe**

WSDL.EXE Flag	Mô tả
/l[language]:	Cho biết ngôn ngữ lập trình sử dụng đến đối với lớp proxy được kết sinh. Bạn có thể khai báo CS (mặc nhiên), VB hoặc JS như là đối mục ngôn ngữ.
/n[namespace]:	Cho khai báo namespace đối với proxy hoặc template được kết sinh. Namespace mặc nhiên là global namespace.
/out:	Cho biết tập tin cất trữ đoạn mã proxy. Trình tiện ích sẽ dẫn xuất tên tập tin mặc nhiên từ tên Web Service, và cất trữ dataset được kết sinh trên nhiều tập tin khác nhau.
/protocol:	Cho biết nghi thức nào phải thi công (mặc nhiên là SOAP). Bạn có thể khai báo SOAP, HttpGet, HttpPost hoặc custom protocol được khai báo trong tập tin config.

### 9.4.1.1 Quan sát Proxy Code



```

using System.Diagnostics;
using System.Xml.Serialization;
using System;
using System.Web.Services.Protocols;
using System.ComponentModel;
using System.Web.Services;

[System.Web.Services.WebServiceBindingAttribute(Name="Service1Soap",
Namespace="http://tempuri.org/")]
public class Service1:
    System.Web.Services.Protocols.SoapHttpClientProtocol
{
    public Service1()
    {
        this.Url = "http://localhost/calculwebsevice/service1.asmx";
    }
}

```

Như bạn có thể thấy, hàm constructor của lớp proxy này duy trì địa chỉ URL của Web Service nằm ở xa và trữ nó trong thuộc tính kế thừa **Url**. Ngoài ra, bạn để ý lớp cơ bản trừu tượng thuộc kiểu dữ liệu **SoapHttpClientProtocol**. Kiểu dữ liệu này khai báo phần lớn thi công đối với việc truyền tin với một SOAP Web Service trên HTTP. Bảng 9-10 mô tả một vài thành viên được kế thừa đáng lưu ý.

**Bảng 9-10: Các thành viên cốt lõi được kế thừa**

Các thành viên	Mô tả
<b>BeginInvoke()</b>	Bắt đầu một triệu gọi hàm bất đồng bộ của một SOAP Web Service.
<b>EndInvoke()</b>	Kết thúc một triệu gọi hàm bất đồng bộ của một SOAP Web Service.
<b>Invoke()</b>	Triệu gọi đồng bộ một hàm hành sự của một SOAP Web Service.
<b>Proxy</b>	Đi lấy hoặc đặt để thông tin proxy để hình thành một Web Service request xuyên qua một bức tường lửa (firewall).
<b>Timeout</b>	Đi lấy hoặc đặt để thời gian đáo hạn (timeout) dùng trong các triệu gọi hàm bất đồng bộ.
<b>Url</b>	Đi lấy hoặc đặt để địa chỉ cơ bản URL đối với server dùng trong các request.
<b>UserAgent</b>	Đi lấy hoặc đặt để trị đối với user agent header được gửi đi với mỗi request.

Lẽ dĩ nhiên “món ăn chính” của proxy được kết sinh là bản thân thi công những hàm hành sự. Đoạn mã được kết sinh của proxy sẽ định nghĩa những thành viên bất đồng bộ và đồng bộ đối với mỗi hàm hành sự được định nghĩa trong Web Service. Sau đây là phần thi công hàm hành sự **Add()**:

```

[System.Web.Services.Protocols.SoapDocumentMethodAttribute("http://tempuri.org/Add", RequestNamespace="http://tempuri.org/",
ResponseNamespace="http://tempuri.org/",
Use=System.Web.Services.Description.SoapBindingUse.Literal,
ParameterStyle=System.Web.Services.Protocols.SoapParameterStyle.Wrapped)]

```

```
public int Add(int x, int y)
{
    object[] results = this.Invoke("Add", new object[] {x, y});
    return ((int) (results[0]));
}
```

Mỗi hàm hành sự Web đều được đánh dấu bởi attribute `SoapMethod`. Ngoài ra, bạn để ý là hàm hành sự `Add()` có cùng dấu ấn như với hàm hành sự Web nguyên thủy. Đối với khách hàng khi gọi `Add()`, phần lô gic sẽ được thi hành trực tiếp. Lẽ dĩ nhiên là trong thức tế các thông số incoming (kèm theo hàm hành sự có mang tên) sẽ được gửi cho **`SoapHttpClientProtocol.Invoke()`**. Tới lúc này, HTTP request được gửi cho đúng địa chỉ URL.

Một nhậ tu bạn phải thực hiện bằng tay đối với tập tin proxy được kết sinh là cho gói lớp này vào trong một định nghĩa namespace như `TheCalcProxy` chẳng hạn, trừ phi bạn khai báo rõ ra một flag **/n: TheCalcProxy** trên dòng lệnh command line:

```
wsdl.exe /n:TheCalcProxy /out:calcproxy.cs http:/...
```

thì lúc này bạn sẽ có:

```
namespace TheCalcProxy
{
    public class Service1:
        System.Web.Services.Protocols.SoapHttpClientProtocol
    {
        . . .
    }
}
```

## 9.4.2 Xây dựng Assembly

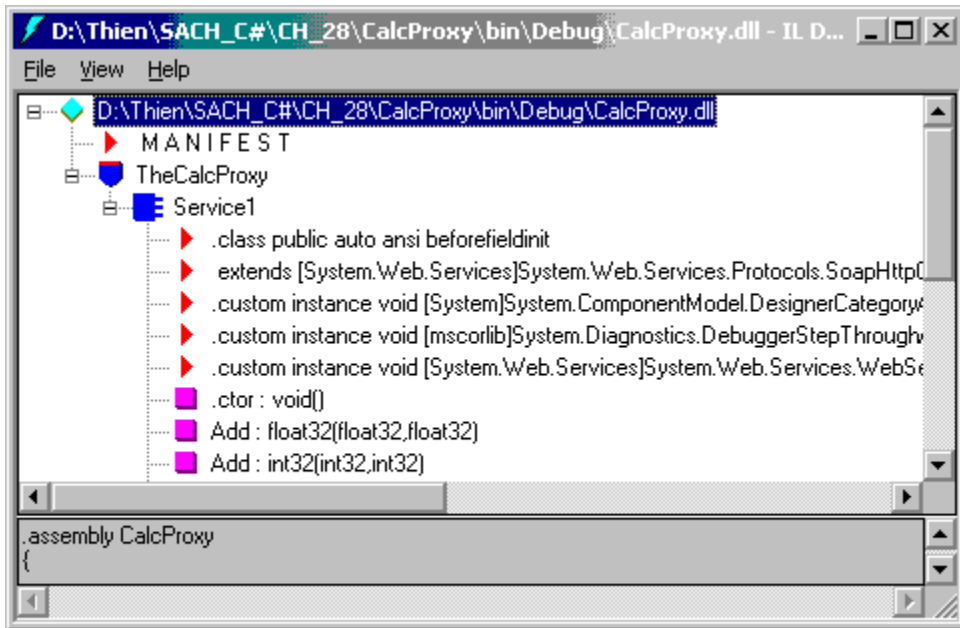
Trước khi bạn có thể tạo những ứng dụng client sử dụng Web Service, bạn cần tạo một assembly chứa kiểu dữ liệu proxy. Bạn có thể sử dụng trực tiếp C# compiler (`csc.exe`) như sau:

```
csc /r:system.web.services.dll /r:system.xml.dll
    /out:d:\thien\calcproxy.dll /t:library d:\thien\calcproxy.cs
```

hoặc chọn một **C# Code Library** khác sử dụng Visual Studio .NET. Bạn vào IDE, ra lệnh **File | New | Project** rồi chọn **Class Library**, cho đặt tên `CalcProxy`, rồi ấn **<OK>**. Bạn trở về dự án mới thiết lập, rồi ra lệnh **Project | Add Existing Item** để cho hiện lên khung đối thoại mở tập tin proxy `CalcProxy.cs`. Cuối cùng, bạn ra lệnh **Project | Add**

Reference để cho hiện lên khung đối thoại Add Reference. Bạn chọn qui chiếu về **System.Web.Services.dll** và **System.Xml.dll**

Kết quả cuối cùng là bạn có một thư viện mới chứa proxy của bạn (hình 9-12).



Hình 9-12: Proxy TheCalcProxy được gói ghém trong một assembly

### 9.4.3 Xây dựng một ứng dụng Client

Tới đây, bạn có thể tạo một ứng dụng kiểu console, kiểu Windows Form hoặc kiểu ASP.NET. Bạn thêm qui chiếu về assembly CalcProxy.dll rồi viết một đoạn mã làm gì đó. Trong mỗi trường hợp, bạn phải bảo đảm qui chiếu về **System.Web.Services.dll** và về TheCalcProxy. Thí dụ, ta chọn ứng dụng console như sau:

```
using System;
using TheCalcProxy;

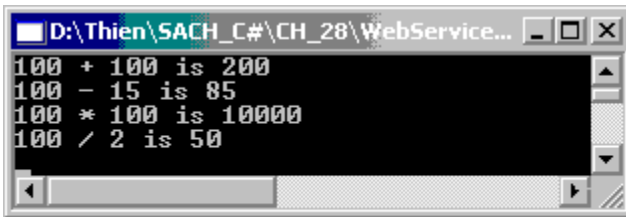
namespace WebServiceConsumer
{
    class WebConsumer
    {
        public static int Main(string[] args)
        {
            Service1 w = new Service1();
            Console.WriteLine("100 + 100 is {0}", w.Add(100, 100));
            Console.WriteLine("100 - 15 is {0}", w.Subtract(100, 15));
            Console.WriteLine("100 * 100 is {0}", w.Multiply(100,
```

```

100));
Console.WriteLine("100 / 2 is {0}", w.Divide(100, 2));

try
{
    w.Divide(0, 0);
}
catch (DivideByZeroException e)
{
    Console.WriteLine(e.Message);
}
Console.ReadLine();
return 0;
}
}
}

```



Hình 9-13: Một console Web Service consumer

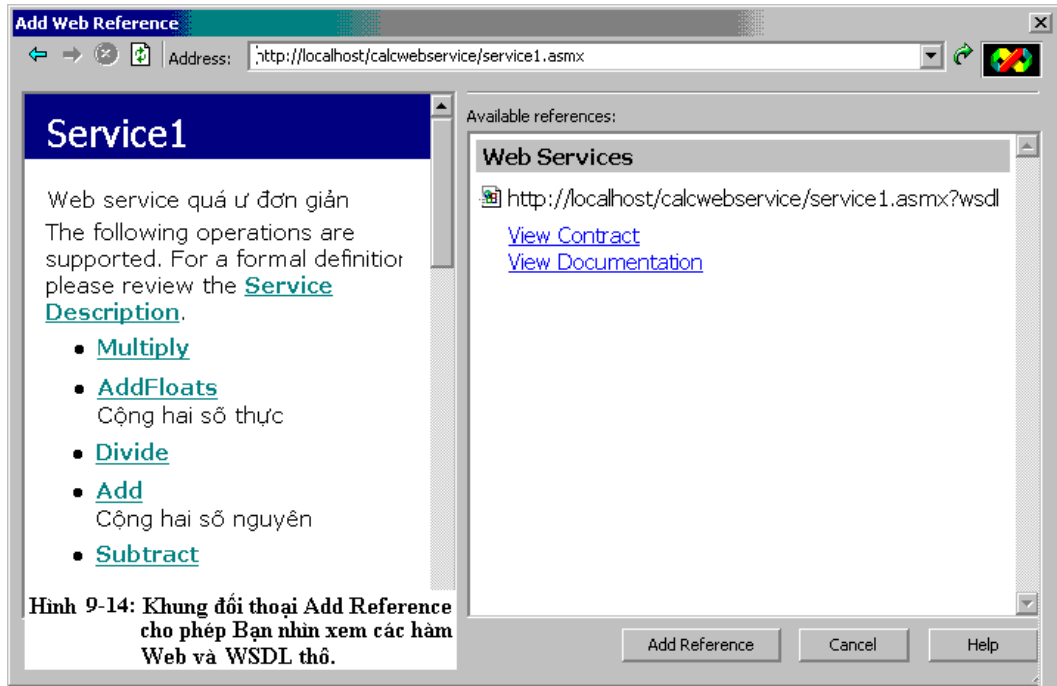
Nếu bạn comment out khối try/catch, thì hình 9-13 cho thấy kết quả. Nhưng nếu bạn thêm khối try/catch, thì một exception sẽ hiện lên do việc chia bởi zero.

## 9.5 Kết sinh một Proxy thông qua VS.NET

Sử dụng trình tiện ích **wsdl.exe** và C# compiler (**csc.exe**) có vẻ hơi lộn cộn. Lợi điểm duy nhất khi sử dụng **wsdl.exe** là trình tiện ích này cho phép bạn trực tiếp khai báo một wire protocol nào đó (GET, POST hoặc SOAP) sử dụng đến flag **/protocol**. Ngược lại, Visual Studio .NET chỉ tạo proxy đáp ứng nghi thức SOAP (tối tân hơn).

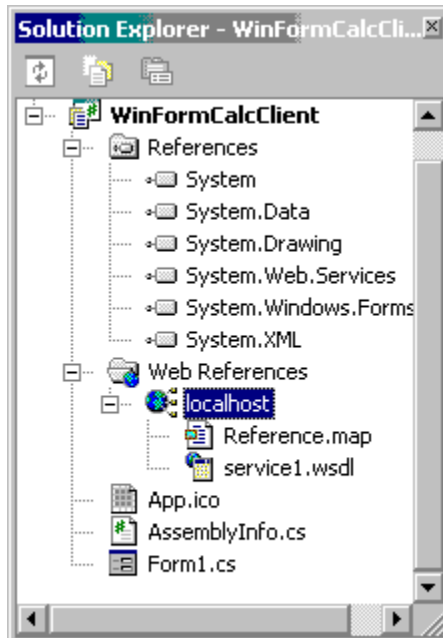
Để minh họa, ta thử xây dựng một ứng dụng client dạng Windows Forms, mang tên WinFormCalcClient chẳng hạn. Giả sử bạn chỉ có một giao diện người sử dụng định nghĩa hai trị được trao qua cho các hàm hành sự **Add()**, **Substract()**, v.v.. Bạn dùng hai ô text box (txtNumb1 và txtNumb2) để nhận hai trị x và y, và 4 Button dành cho 4 phép toán (btnAdd, btnSubtract, btnMultiply và btnDivide) và cuối cùng một label nhận kết quả (lblAns). Một khi bạn đã tạo xong GUI, bạn có thể thêm một Web reference vào dự án của mình. Bạn có thể khở vào URL chỉ về tập tin \*.asmx (ở đây là <http://localhost/CalcWebService/Service1.asmx>) đối với dự án của mình. Hình 9-14 cho thấy khế ước WSDL và nhóm các hàm hành sự Web.

Bây giờ, bạn chọn **View Contract** và ấn nút **<Add Reference>** trên hình 9-14. Một mắt gút **Web Reference** được thêm vào Solution Explorer của dự án. Xem hình 9-15.



Hình 9-14: Khung đối thoại Add Reference cho phép Bạn nhìn xem các hàm Web và WSDL thố.

Tới đây coi như bạn sẵn sàng khai báo công việc sẽ được thực hiện trực tiếp với Service1. Bạn để ý là namespace được kết sinh là tên của máy tính chịu lưu trữ Web Service, ở đây là localhost. Bạn thử tạo hàm thụ lý tình huống khi ấn nút <Add>:

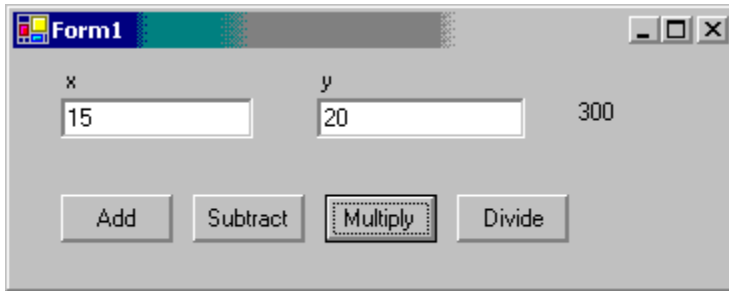


Hình 9-15: Web Reference node

```
using localhost;
public class MainForm
    : System.Windows.Forms.Form

private void btnAdd_Click(object sender,
    System.EventArgs e)
{
    localhost.Service1 w = new
        localhost.Service1();
    int ans = w.Add(
        int.Parse(txtNum1.Text),
        int.Parse(txtNum2.Text));
    lblAns.Text = ans.ToString();
}
```

Với các nút btnSubtract, btnMultiply và btnDivide bạn cũng viết đoạn mã thụ lý tình huống như trên. Bạn cho Build rồi chạy Debug. Bạn thấy kết quả như theo hình 9-16.



Hình 9-16: Kết quả sử dụng CalcWebService

## 9.6 Một Web Service lý thú hơn cùng với Web Client

Sức mạnh thực sự của Web Service nằm trong việc khi bạn xây dựng những hàm hành sự Web trả về những kiểu dữ liệu phức tạp, không như thí dụ quá đơn giản CalcWebService mà chúng ta vừa xem qua. Do đó, ta nên xây dựng một Web Service hấp dẫn hơn có thể trả về DataSet của ADO.NET, bản đây các kiểu dữ liệu hoặc kiểu dữ liệu custom. Như bạn đã biết ta cần dùng đến SOAP.

Bạn cho tạo một dự án C# ASP.NET Web Service, mang tên **CarsWebService**. Mục đích đầu tiên là tạo một hàm hành sự Web trả về một DataSet thuộc ADO.NET, chứa các mẫu tin của bảng dữ liệu Inventory trên căn cứ dữ liệu Cars.mdb. Trị trả về lẽ dĩ nhiên là một DataSet. Phần thi công logic sẽ điền DataSet sử dụng OleDbDataAdapter như sau đây:

```
// Trả về tất cả các xe trên bảng dữ liệu Inventory
[WebMethod]
public DataSet GetAllCars()
{
    // Điền DataGridView với dữ liệu từ bảng dữ liệu Inventory
    OleDbConnection conn = new OleDbConnection();
    conn.ConnectionString = "Provider=Microsoft.Jet.OLEDB.4.0;"+
        "Data Source=D:\\Thien\\Cars.mdb";
    OleDbDataAdapter da = new OleDbDataAdapter(
        "Select * from Inventory", conn);
    DataSet ds = new DataSet();
    da.Fill(ds, "Inventory");
    return ds;
}
```

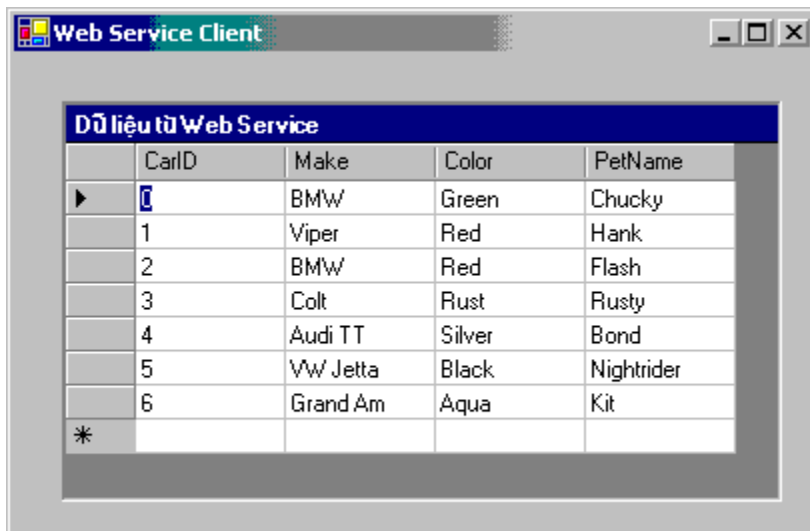
Bạn nhớ thêm chỉ thị using **System.Data.OleDb** vào đầu dự án.

Tiếp theo, bạn tạo một ứng dụng Windows Form client, mang tên

**WebServiceClient**, thêm một qui chiếu về Web Service, rồi triệu gọi hàm **GetAllCars()** trong hàm tình huống **Form\_Load** và gắn kết bảng dữ liệu Inventory vào một ô control **DataGrid**, như theo sau đây:

```
private void Form1_Load(object sender, System.EventArgs e)
{
    localhost.Service1 s = new localhost.Service1();
    DataSet ds = s.GetAllCars();
    dataGrid1.DataSource = ds.Tables["Inventory"];
}
```

Bạn nhớ thêm chỉ thị using localhost vào đầu lớp Form1. Hình 9-17 cho thấy kết quả chạy WebServiceClient.



Hình 9-17: Nhận được DataSet từ CarWebService.

## 9.6.1 Sản sinh hằng loạt kiểu dữ liệu custom

Nghi thức SOAP cũng có khả năng chuyển kiểu dữ liệu custom được biểu diễn theo XML. Để minh họa, ta thử xây dựng kiểu dữ liệu Car cuối cùng. Bây giờ ta thử chèn vào một lớp mang tên Car vào ứng dụng CarsWebService hiện hành.

```
using System;
using System.Collections;
using System.ComponentModel;
using System.Data;
using System.Diagnostics;
using System.Web;
using System.Web.Services;
using System.Data.OleDb;
```

```

using System.Xml.Serialization;

namespace CarsWebService
{
    public class Service1: System.Web.Services.WebService
    {
        public Service1()
        {
            InitializeComponent();
        }
        . . .
        [WebMethod]
        public DataSet GetAllCars()
        {
            OleDbConnection conn = new OleDbConnection();
            conn.ConnectionString =
                "Provider=Microsoft.Jet.OLEDB.4.0;" +
                "Data Source=D:\\Thien\\Cars.mdb";
            OleDbDataAdapter dad = new OleDbDataAdapter(
                "Select * from Inventory", conn);
            DataSet dst = new DataSet();
            dad.Fill(dst, "Inventory");
            return ds;
        }
    }

    [XmlInclude(typeof(Car))]
    public class Car
    {
        public Car(){}
        public Car(string n, int s)
        {
            petName = n;
            maxSpeed = s;
        }
        public string petName;
        public int maxSpeed;
    }
}

```

Về mặt kỹ thuật, tiến trình biến đổi dữ liệu có mang trạng thái (stateful data) của một đối tượng thành một biểu diễn XML được gọi là *serialization* (sản sinh hằng loạt). Vì bản thân XML không cho biết là kiểu xe nào, bạn phải cho đánh dấu mỗi kiểu dữ liệu custom cần được serialize như là XML bằng cách dùng thuộc tính `XmlIncludeAttribute`.

Tiếp theo, bạn cho định nghĩa một biến thành viên private kiểu dữ liệu `ArrayList`, mang tên `carList`, rồi cho điền bản dãy này với một số xe ban đầu trong hàm constructor của Web Service, như sau:

```

private ArrayList carList = new ArrayList();

public Service1()
{
    InitializeComponent();
}

```



```
// Thêm vài xe lúc ban đầu
carList.Add(new Car("Zippy", 170));
carList.Add(new Car("Fred", 80));
carList.Add(new Car("Sally", 40));
}
```

Bây giờ, bạn thêm hai hàm hành sự Web, `GetCarList()` trả về trọn bản dãy các xe, và `GetACarFromList()` trả về một xe nào đó lấy từ bản dãy dựa trên chỉ số. Sau đây là phần thi công của hai hàm hành sự này:

```
// Trả về một xe nào đó từ danh sách
[WebMethod]
public Car GetACarFromList(int carToGet)
{
    if(carToGet <= carList.Count)
    {
        return (Car) carList[carToGet];
    }
    throw new IndexOutOfRangeException();
}

// Trả về toàn bộ các xe
[WebMethod]
public ArrayList GetCarList()
{
    return carList;
}
```

Hình 9-18 cho thấy `CarsWebService` có bao nhiêu hàm hành sự Web.

## 9.6.2 Cho biểu mẫu thêm phong phú

Bước kế tiếp là làm sao sử dụng hai hàm hành sự vừa mới thêm vào **`GetCarList()`** và **`GetACarFromList()`** từ ứng dụng client.

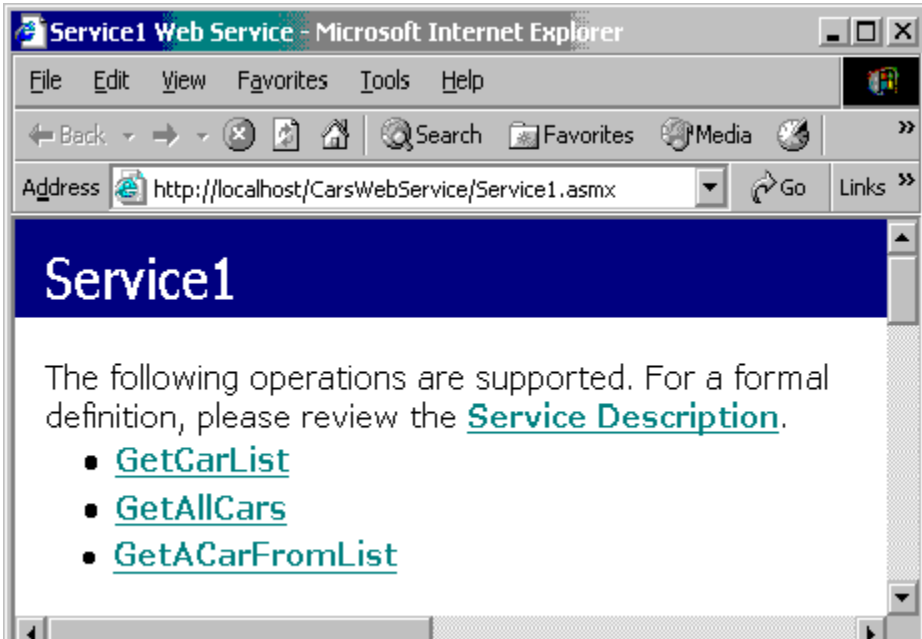
Vì Web Service là một assembly, bạn nên hiểu kiểu dữ liệu `Car` sẽ được mô tả trong metadata của assembly. Bạn cho qui chiếu lại `CarsWebService` để nhậ tu hai hàm hành sự bạn mới thêm vào. Để minh họa, bạn có thể gọi ra một xe nào đó như sau, với nút <Get Car From List>:

```
private void btnGetCar_Click(object sender, System.EventArgs e)
{
    try
    {
        localhost.Car c;
        localhost.Service1 cws = new localhost.Service1();
        c = cws.GetACarFromList(int.Parse(txtCarToGet.Text));
        MessageBox.Show(c.petName, "Car " + txtCarToGet.Text
            + " mang tên:");
        cws.Dispose();
    }
}
```

```

catch
{
    MessageBox.Show("Không xe nào với số này...");
}
}

```



Hình 9-18: Web Service CarsWebService

Hình 9-19 cho kết quả trắc nghiệm nút <Get Car From List>.

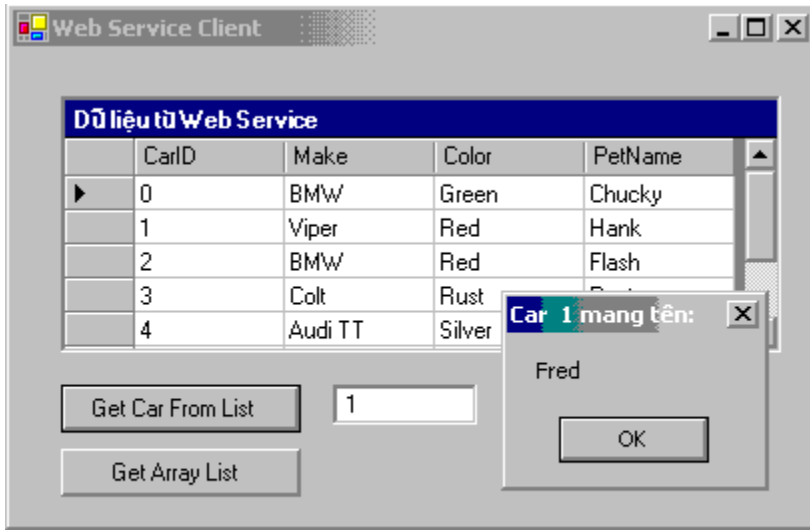
Còn đối với ArrayList được trả về bởi hàm GetCarList, bạn có thể cho giữ lại trọn tập hợp các items trên một bản dãy đối tượng và cho ép kiểu một cách thích ứng, như sau:

```

private void btnArrayList_Click(object sender, System.EventArgs e)
{
    localhost.Service1 cws = new localhost.Service1();
    object[] objs = cws.GetCarList();
    string petNames = "";

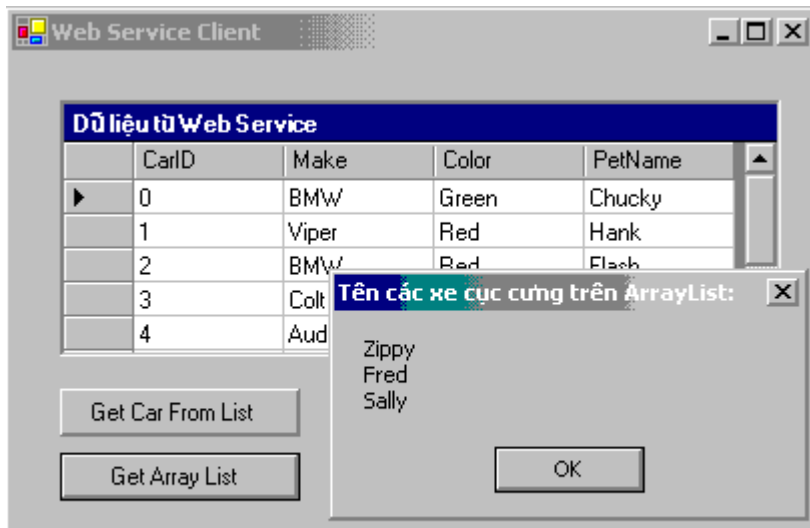
    // In ra pet name cho mỗi item trong bản dãy
    for(int i=0; i < objs.Length; i++)
    {
        // Trích xe kế tiếp
        localhost.Car c = (localhost.Car)objs[i];
        petNames += c.petName + "\n";
    }
    MessageBox.Show(petNames, "Tên các xe cục cưng trên ArrayList:");
    cws.Dispose();
}

```



Hình 9-19: Lấy ra một xe từ ArrayList

Hình 9-20 cho thấy kết quả trắc nghiệm nút <Get Array List>:



Hình 9-20: Trắc nghiệm nút Get Array List

### 9.6.3 Xây dựng các kiểu dữ liệu serializable

Khi một kiểu dữ liệu custom được serialized thành XML, thực chất bạn cho trữ dữ liệu có mang trạng thái từ đối tượng để dùng về sau. Do đó, muốn serialize một biến

thành viên nào đó vào lúc chạy, CLR Runtime phải có cách truy cập trị nằm ẩn phía sau. Như bạn còn nhớ, lớp Car bạn đã định nghĩa trước đây, chịu hỗ trợ hai vùng mục tin public, như sau:

```
[XmlInclude(typeof(Car))]  
public class Car  
{  
    public Car(){}  
    public Car(string n, int s)  
    {  
        petName = n;  
        maxSpeed = s;  
    }  
    public string petName;  
    public int maxSpeed;  
}
```

Bạn có thể thay đổi các biến thành viên này thành private như sau:

```
[XmlInclude(typeof(Car))]  
public class Car  
{  
    public Car(){}  
    public Car(string n, int s)  
    {  
        petName = n;  
        maxSpeed = s;  
    }  
  
    // Có serialize không ?  
    private string petName;  
    private int maxSpeed;  
}
```

Khi bạn trắc nghiệm hàm hành sự GetCarList() lại một lần nữa, bạn sẽ thấy một ArrayList với 3 xe. Tuy nhiên, mỗi item sẽ không có state data được biểu diễn trong XML. Lý do rất đơn giản: CLR Runtime không thể truy cập dữ liệu private. Do đó, muốn trạng thái của kiểu dữ liệu của bạn cần được serialize thành XML bạn phải hoặc cung cấp các vùng mục tin public, hoặc cung cấp một cách truy cập các vùng mục tin private sử dụng một thuộc tính tương ứng, như sau:

```
[XmlInclude(typeof(Car))]  
public class Car  
{  
    . . .  
    private string petName;  
    private int maxSpeed;  
  
    public string PetName  
    {  
        get {return petName;}  
        set {petName = value;}  
    }  
  
    public string MaxSpeed
```

```
{    get {return maxSpeed;}  
    set {maxSpeed = value;}  
}  
}
```

Nói tóm lại, xây dựng những kiểu dữ liệu có sẵn trên Web Service ít nhiều cũng giống như xây dựng những kiểu dữ liệu C#. Các điểm thực sự phải quan tâm là bạn cần đánh dấu kiểu dữ liệu với thuộc tính [XMLInclude] và cho về public những dữ liệu nào muốn được biểu diễn theo XML.

## 9.7 Tìm hiểu Discovery Service Protocol

Đề mục cuối cùng của chương này là tìm hiểu tập tin mang tên \*.vsdisco. Bất cứ lúc nào một client ở xa (hoặc cục bộ) muốn sử dụng một Web Service, thì bước đầu tiên là xác định Web Service nào hiện hữu trên một máy tính nào đó. Trong khi thư viện các lớp .NET định nghĩa những kiểu dữ liệu cho phép bạn khảo sát những Web Service được đăng ký về mặt lập trình, dịch vụ khám phá (discovery service) cũng được đòi hỏi bởi nhiều công cụ CASE vào lúc thiết kế (chẳng hạn Add Web Reference Wizard).

Tập tin \*.vsdisco dùng mô tả mỗi Web Service trên một thư mục ảo và trên bất cứ thư mục con có liên hệ. Khi bạn tạo một Visual Studio .NET Web project, bạn tự động nhận được một tập tin \*.vsdisco giống như sau:

```
<?xml version="1.0" encoding="utf-8" ?>  
<dynamicDiscovery xmlns=  
    "urn:schemas-dynamicdiscovery:disco.2000-03-17">  
    <exclude path="_vti_cnf" />  
    <exclude path="_vti_pvt" />  
    <exclude path="_vti_log" />  
    <exclude path="_vti_script" />  
    <exclude path="_vti_txt" />  
    <exclude path="Web References" />  
</dynamicDiscovery>
```

Tag **dynamicDiscovery** cho biết tập tin \*.vsdisco phải được xử lý trên server để trả về một mô tả XML đối với mỗi Web Service trên một thư mục ảo nào đó. Ngoài tag kể trên, bạn còn thấy một số các lỗi tìm về (path) không có ý nghĩa khỏi phải truy tìm. Để minh họa bạn cho khởi động Internet Explorer rồi lái về tập tin CarsWebService.vsdisco. Nội dung giống như trên.

### 9.7.1 Thêm một Web Service mới

Bạn nên hiểu là một tập tin \*.vsdisco đơn độc mô tả mỗi một Web Service được cài đặt tại một thư mục ảo nào đó. Giả sử bây giờ bạn muốn thêm một Web Service khác vào

dự án CarsWebService hiện hành bằng cách ra lệnh **Project | Add Web Service...** Và khi khung đối thoại **Add New Item** hiện lên, bạn chọn Web Service, và kho vào **MotorBikes.asmx** rồi ấn <Open> thì một tập tin MotorBikes.asmx được mở ra. Bạn định nghĩa một hàm hành sự Web **GetBikerDesc** duy nhất như sau:

```
[WebMethod]
public string GetBikerDesc()
{   return "Name: Tiny. Weight: 374 pounds";
}
```

Nếu bạn cho biên dịch lại ứng dụng và nhìn lại xem \*.vsdisco từ IE. Bạn sẽ thấy giờ đây có hai mắt gút <contractRef> một cho Service1.asmx và một cho MotorBikes.asmx. Thật ra xem kết quả của một disco query từ trong lòng IE không có chi là hấp dẫn. Tuy nhiên, bạn nhớ lại cũng tập tin này được dùng với Add Web Reference Wizard. Ngoài ra, bạn cũng có thể thông qua lập trình có được cũng những thông tin này sử dụng các kiểu dữ liệu thiên về WSDL. Bạn nên tìm hiểu về đề tài này.

Như đã nói trong chương trước, chương này cũng chỉ mới phác họa sơ về Web Services. Muốn tìm hiểu sâu đề tài này, đề nghị bạn tìm đọc tập V bộ sách .NET Toàn tập này khi nói về lập trình Web với ASP.NET và C#

## Bản Chỉ Mục

\*.aspx, khảo sát tập tin, 515  
\*.aspx/Codebehind Connection, 520  
.NET to COM Interoperability, 404  
.Net Type, cho đăng ký, 456  
/addModules, Switch của csc.exe, 253  
Accept(), hàm, 78  
Accept()., 77  
AcceptButton, 153  
Activate(), 154  
Activated, 154  
Activator, Lớp, Kết nối trễ (late binding), 308  
Active Server Page cổ điển, 504  
ActiveControl, thuộc tính, Lớp Container Control, 152  
ActiveMDIChild, 153  
ActiveX control, Sử dụng ActiveX control trên.NET, 421  
ActiveX control, Tạo một..., 422  
Add Reference, cửa sổ, 120  
Add, 158  
AddMessageFilter(),Lớp Application, 126  
AddMessageFilter,Lớp Application, 131  
AddRange, 158  
AddValue(),Lớp SerializationInfo, 110  
AdRotator Control, 537  
AdRotator Control, Làm việc với..., 539  
Alignment, thuộc tính lớp StatusBarPanel, 171  
AllKeys, thuộc tính  
HttpApplicationState, 525

Alt, 147  
AppDomain, Thí dụ sử dụng, 347  
Appearance, thuộc tính lớp ToolBar, 177  
AppendHeader, hàm HttpResponseMessage, 524  
AppendText(), 42  
AppendText(),Lớp FileInfo, 49  
AppendToLog(), hàm HttpResponseMessage, 524  
Application configuration file, 262  
Application directory, 260  
Application, 118  
Application, Lớp, Các hàm cốt lõi của..., 126  
Application, Lớp, Các tình huống, 127  
Application, Lớp, Thực hành với ..., 128  
Application, Lớp, Tìm hiểu, 126  
Application, thuộc tính Page, 519  
Application/Session, Tìm hiểu sự phân biệt, 511  
ApplicationExit, Lớp Application, 127  
ApplicationExit, Tình huống, 129  
ApplicationPath, lớp HttpRequest, 522  
ASP .NET Namespaces, 510  
Assembly Manifest, 232  
Assembly reference (gọi tắt AsmRef)., 264  
Assembly Registration, Tương tác với, 468  
Assembly, 229

Assembly, Cấu trúc của một assembly, 230  
 Assembly, Phiên bản hóa một assembly, 237  
 Assembly, Tổng quan, 229  
 Assembly, Thử nghiệm, 257  
 Assembly.GetType(), 302  
 Assembly.Load(), 300  
 AssemblyBuilder, 312  
 AssemblyBuilderAccess, Enumeration, 315  
 AssemblyInfo.cs, tập tin, 290  
 AssemblyLoad, tình huống lớp  
 AppDomain, 345  
 AssemblyResolve, tình huống lớp  
 AppDomain, 345  
 AssemblyResolver, Nạp assembly  
 vào ứng dụng, 258  
 Asynchronous I/O, 72  
 Asynchronous Network File  
 Streaming, 85  
 asynchronous streaming network  
 client, Thiết đặt một..., 84  
 asynchronous streaming network file  
 server, Thiết đặt một..., 88  
 ATL COM server, xây dựng một ...,  
 428  
 Attribute, Áp dụng các ..., 281  
 Attribute, Attribute target, 280  
 Attribute, AttributeUsage, 284  
 Attribute, custom attribute, 280  
 Attribute, Custom attribute, 283  
 Attribute, Đặt tên cho một attribute,  
 284  
 Attribute, intrinsic attribute, 280  
 Attribute, Khai báo một attribute, 284  
 Attribute, Khám phá các attribute vào  
 lúc chạy, 291  
 Attribute, Sử dụng một attribute, 285

Attribute, Xây dựng một attribuet,  
 285  
 Attributes, 30  
 Attributes, Tìm hiểu về..., 279  
 AttributeTargets, Enumeration, 280  
 AutoFlush(), Thuộc tính, 62  
 AutoPostBack, attribute, 484  
 AutoScale, 153  
 AutoScroll, 151  
 AutoScrollMinSize, 151  
 AutoSize, thuộc tính lớp  
 StatusBarPanel, 171  
 AutoSize, thuộc tính lớp ToolBar,  
 177  
 AxImp.exe, 421  
 BackgroundImage, 170  
 BaseStream, 70  
 BeginRead(), 54, 72  
 BeginWrite(), 54  
 Biểu mẫu, Cho đặt đề style của  
 một..., 137  
 Binary Reader, BinaryWriter, Lớp,  
 28  
 BinaryFormatter, Formatter, 96  
 BinaryFormatter, Sử dụng lớp..., 98  
 BinaryReader, Các thành viên của ...,  
 70  
 BinaryReader, Lớp, 69  
 BinaryWriter, Lớp, 69  
 BinaryWriter, Lớp, Các thành viên  
 của ..., 69  
 BorderStyle, 153  
 BorderStyle, thuộc tính lớp  
 StatusBarPanel, 171  
 BorderStyle, thuộc tính lớp ToolBar,  
 177  
 Bound, 141  
 Bound, thuộc tính, 141  
 Browser, lớp HttpRequest, 522



BufferedStream, Làm việc với..., 59  
 BufferedStream, Lớp, 28  
 Buttons, thuộc tính lớp ToolBar, 177  
 ButtonSize, thuộc tính lớp ToolBar, 177  
 C# Web Application, Tạo một, 513  
 Cache, thuộc tính HttpResponseMessage, 523  
 Cache, thuộc tính Page, 519  
 Cái đếm qui chiếu (reference count) của coclass, 406  
 Calendar Control, 537  
 Calendar Control, Làm việc, 537  
 Callback, hàm hành sự, 72  
 CallingConvention, vùng mục tin của DllImportAttribute, 400  
 CallingConventions, vùng mục tin của DllImportAttribute, 402  
 CancelButton, 153  
 CanRead, 54  
 CanSeek, 54  
 Capacity, 58  
 CCW (COM Callable Wrapper), 453  
 CenterToScreen(), 154  
 Change, tình huống, 485  
 ChannelServices, Lớp, 360  
 CharacterSet, vùng mục tin của DllImportAttribute, 402  
 CharSet, Enumeration, 402  
 CharSet, vùng mục tin của DllImportAttribute, 400  
 Checked, Lớp MenuItem, 164  
 Chết chum (deadlock), 394  
 Class Interface, Định nghĩa, 455  
 Class Interface, Tìm hiểu..., 454  
 ClassInterfaceAttribute, 455  
 ClassInterfaceAttribute, Lớp Runtime.InteropServices, 398  
 ClassInterfaceType, enumeration, 455

Clear(), 158  
 Clear(), hàm HttpResponseMessage, 524  
 Client, Xây dựng một client, 362  
 Client-activated, đối tượng, 358  
 Client-Side Scripting, 498  
 ClipRectangle, Lớp PaintEventArgs, 151  
 CloneMenu, Lớp Menu, 157  
 Close(), 154  
 Close(), 54, 61, 70  
 Close(), hàm HttpResponseMessage, 524  
 Close(), Lớp RegistryKey, 212  
 Closed, 154  
 Closing, 154  
 Coclass (và COM Properties)  
 Conversion, 443  
 Codebehind, attribute, 518  
 Code-behind, tập tin, 482  
 COFF (Common Object File Format), 230  
 COM Client VB, tạo một, 411  
 COM Enumeration Conversion, 446  
 COM Error, cho tung ra một..., 433  
 COM Event, cho phát pháo một, 431  
 COM Interface Conversion, 440  
 COM SAFEARRAY, Làm việc với..., 447  
 COM server VB, tạo một, 407  
 COM Type như, 405  
 COM type tương đương với kiểu dữ liệu .NET, 405  
 COM Type, như là .NET tương đương, 405  
 COM+ Services, Tương tác, 469  
 COM+, Xây dựng kiểu dữ liệu chịu chơi với..., 472  
 COMInterfaceType, Enumeration, 441  
 Common Dialog Boxes., 118

CommonAppDataRegistry, Lớp Application, 127  
 CompanyName, Lớp Application, 127  
 CompareValidator, 544  
 Compatibility version, Version identifier, 237  
 Component Service Explorer, 476  
 Component, Lớp, 133  
 Components, 117  
 ComRegisterFunction Attribute, 468  
 ComRegisterFunctionAttribute, Lớp Runtime.InteropServices, 398  
 ComSourceInterfacesAttribute, Lớp Runtime.InteropServices, 398  
 ComUnregisterFunctionAttribute, 468  
 ComUnregisterFunctionAttribute, Lớp Runtime.InteropServices, 398  
 ConstructorBuilder, 312  
 ContainerControl, Lớp, 152  
 ContainerControl, Lớp, Các thành viên của..., 152  
 Contains(), 158  
 ContentEncoding, thuộc tính HttpResponseMessage, 523  
 ContentType, lớp HttpRequest, 522  
 ContentType, thuộc tính HttpResponseMessage, 523  
 ContextMenu, Lớp, 162  
 Control, 147  
 Control, Lớp cơ bản, Các thuộc tính, 532  
 Control, Lớp, 117, 135  
 Control, Lớp, Các hàm hành sự chủ chốt, 136  
 Control, Lớp, Các hàm hành sự của..., 149

Control, Lớp, Các thuộc tính chủ chốt, 136  
 Control, Lớp, Các thuộc tính của..., 148  
 Control, Lớp, Các tình huống chủ yếu, 141  
 ControlBox, 153  
 ControlStyles, Enumeration, 137  
 Cookies, lớp HttpRequest, 522  
 Cookies, thuộc tính HttpResponseMessage, 523  
 CopyTo(), 42  
 CopyTo(), Lớp FileInfo, 51  
 Count, 158  
 Count, thuộc tính  
 HttpApplicationState, 525  
 Create(), 31, 42  
 CreateChildControl(), hàm, 486  
 CreateChildControls(), hàm, 485  
 CreateComInstanceFrom, hàm, Lớp Activator, 308  
 CreateDomain(), hàm của lớp AppDomain, 344, 347  
 CreateInstance(), hàm lớp Activator, 417  
 CreateInstance, hàm, Lớp Activator, 308  
 CreateInstanceFrom, hàm, Lớp Activator, 308  
 CreateSubdirectory(), 31  
 CreateSubDirectory(), Lớp DirectoryInfo, 50  
 CreateSubdirectory(), tạo thư mục con, 38  
 CreateSubKey(), Lớp RegistryKey, 212  
 CreateText(), 42  
 CreateText(), Lớp FileInfo, 49  
 CreationTime, 30

Cross-language inheritance (kế thừa xuyên ngôn ngữ), 229  
Cross-language inheritance, 244  
Cửa sổ chính, Tạo bằng tay, 120  
CurrentCulture, Lớp Application, 127  
CurrentDomain, thuộc tính lớp AppDomain, 344  
CurrentInputLanguage, Lớp Application, 127  
CurrentThread, thuộc tính lớp Thread, 372  
Cursors, 151  
Custom attribute, 283  
Custom Serialization, 109  
CustomAttributeBuilder, 312  
CustomValidator, 544  
Data Binding, 543  
DataGrid, 541  
DataGrid, Cho điền dữ liệu, 541  
DataList, 541  
Deadlock, 393  
Decrement, lớp Interlocked, 386  
DefaultItem, Lớp MenuItem, 164  
DefineDynamicAssembly(), hàm của lớp AppDomain, 344  
Delegate, 72  
Delete(), 31, 42  
Delete(), Lớp FileSystemInfo, 30  
DeleteSubKey(), Lớp RegistryKey, 212  
DeleteSubKeyTree(), Lớp RegistryKey, 212  
Deserialize(), Lớp, 98  
Điều kiện chạy đua, 393  
Directory, Các thành viên static của lớp, 40  
Directory, DirectoryInfo, Lớp, 28  
Directory, Lớp, 29

DirectoryInfo, Các thành viên, 31  
DirectoryInfo, Lớp, 29  
DirectoryInfo, Lớp., 31  
DirectoryInfo, tạo một đối tượng, 31  
DispIdAttribute, Lớp  
Runtime.InteropServices, 398  
DisplayRectangle(), 156  
Dispose(), hàm., 486  
Disposed, tình huống, 133  
DllImportAttribute, Lớp  
Runtime.InteropServices, 398  
DllImportAttribute, Lớp, Các vùng mục tin của DllImportAttribute, 400  
DllRegisterServer, 468  
DllUnregisterServer, 468  
Đọc và viết dữ liệu, 53  
DoEvents(), Lớp Application, 126  
Đối tượng well-known single-call, 358  
Đối tượng well-known singleton, 358  
Đồng bộ hoá, 382  
Đồng bộ hóa, Sử dụng C# Locks, 387  
DropDownArrows, thuộc tính lớp ToolBar, 177  
DropDownMenu, thuộc tính lớp ToolBarButton, 178  
Dữ liệu nhị phân, Làm việc với..., 69  
Dữ liệu vãng lai, Thụ lý, 103  
Dynamic Assembly, Emitting một, 312  
Dynamic Assembly, Sử dụng thế nào?, 316  
Dynamic assembly, Tìm hiểu và xây dựng dynamic assembly, 311  
Dynamic Invocation, Sử dụng Interfaces, 330

- Dynamic Invocation, Sử dụng
- InvokeMember(), 321
- Dynamic Invocation, Sử dụng
- Reflection Emit, 335
- Early binding (gắn kết sớm)., 414
- Empty Project, 119
- Enabled, Lớp MenuItem, 164
- Enabled, thuộc tính lớp Timer, 174
- Enabled, thuộc tính lớp
- ToolBarButton, 178
- End(), hàm HttpResponseMessage, 524
- Endpoint, 360
- Endpoints, Tìm hiểu..., 366
- EndRead(), Lớp Stream, 74
- Enter(), hàm lớp Monitor, 388
- EntryPoint, vùng mục tin của
- DllImportAttribute, 400, 402
- EntryPointNotFoundException, 401
- EnumBuilder, 312
- Event Viewer, 216
- EventBuilder, 312
- EventLog, Lớp, 217
- EventLog, Lớp, 219
- EventLog, Lớp, Các thành viên, 217
- EventLogEntry, Lớp, 217
- EventLogEntry, Lớp, Các thành viên, 220
- EventLogNames, 217
- ExactSpelling, vùng mục tin của
- DllImportAttribute, 401
- ExecuteAssembly(), hàm của lớp
- AppDomain, 344
- Exit(), ExitThread(), hàm, Lớp
- Application, 126
- Exit(), Lớp Application, 126
- Exits(), 42
- Exits, 30
- ExitThread(), Lớp Application, 126
- Exported Type Information, 456
- Extension, 30
- FieldBuilder, 312
- File | New Project, 119
- File, FileInfo, Lớp, 28
- File, Lớp, 29
- FileAccess, Enumeration, 48
- FileAttributes, Enumeration., 33
- FileInfo, Các thành viên, 42
- FileInfo, Lớp, 29
- FileMode, Enumeration, 47
- FilePath, lớp HttpRequest, 522
- Files, lớp HttpRequest, 522
- FileShare, Enumeration, 48
- FileStream, Làm việc với..., 56
- FileStream, Lớp, 28
- FileStream., 45
- FileSystemInfo, Cá thuộc tính, 30
- FileSystemInfo, Lớp base abstract, 30
- Filter, lớp HttpRequest, 522
- Filter, thuộc tính HttpResponseMessage, 523
- Flush(), 54, 61, 70
- Flush(), hàm HttpResponseMessage, 524
- Font, 170
- ForeColor, 170
- Form, lớp HttpRequest, 522
- Form, Lớp, 153
- Form, Lớp, Các hàm hành sự của..., 154
- Form, Lớp, Các thuộc tính của..., 153
- Form, Lớp, Các tình huống của..., 154
- Format Headers, 490
- Formatter, 343
- Formatter, 97
- Formatter, Sử dụng một..., 96
- FriendlyName, thuộc tính lớp
- AppDomain, 344

FullName, 30  
 gacutil.exe, trình tiện ích, 271  
 gdi32.dll, 399  
 Generated IDL, 466  
 GET & POST, 502  
 GetAssemblies(), hàm của lớp  
 AppDomain, 346  
 GetAttributes(), 42  
 GetBuffer(), 58  
 GetCreationTime(), 42  
 GetCurrentThreadID(), hàm của lớp  
 AppDomain, 344  
 GetCustomAttributes, 299  
 GetData(), hàm của lớp AppDomain,  
 344  
 GetData(), hàm lớp Thread, 372  
 GetDirectories(), 31  
 GetDirectories()., 38  
 GetDomain(), hàm lớp Thread, 372  
 GetDomainID(), hàm lớp Thread,  
 372  
 GetFiles(), 31, 34  
 GetForm(), Lớp MainMenu, 160  
 GetIDsOfNames(), hàm, 416  
 GetLastAccessTime(), 42  
 GetLastWriteTime(), 42  
 GetLogicalDrives()., 40  
 GetMainMenu, Lớp Menu, 157  
 GetMembers(), hàm của lớp Type,  
 303  
 GetMethod()., hàm lớp Type, 306  
 GetMethods(), hàm lớp Type, 306  
 GetObject, hàm, Lớp Activator, 308  
 GetObjectData(), 110  
 GetParameters(), hàm lớp Type, 306  
 GetStream(), 80  
 GetStringBuilder(), 68  
 GetStyle(), hàm, 138

GetSubKeyNames(), Lớp  
 RegistryKey, 212  
 GetType(), hàm, 303  
 GetTypeFromProgID(), hàm lớp  
 Type, 417  
 GetValue(), Lớp RegistryKey, 212  
 GetValueNames(), Lớp RegistryKey,  
 212  
 GetXXXX(), Lớp SerializationInfo,  
 110  
 Giao diện COM bị che dấu bởi  
 RCW, 407  
 Global Assembly Cache (GAC, 265  
 Global.asax, khảo sát tập tin, 516  
 Graphics, Lớp PaintEventArgs, 151  
 GuidAttribute, Lớp  
 Runtime.InteropServices, 398  
 Handle Postback Events, tình huống,  
 485  
 Handle, 157  
 Handle, Lớp Menu, 157  
 Handled, 147  
 Headers, lớp HttpRequest, 522  
 HTML control, liệt kê, 494  
 HTML Form, Triển khai, 492  
 HttpApplicationState, Lớp, 524  
 HttpApplicationState, Lớp, Các  
 thuộc tính cốt lõi, 525  
 HttpMethod, lớp HttpRequest, 522  
 HttpRequest, Lớp, Các thành viên  
 cốt lõi, 522  
 HttpResponse, Lớp, 523  
 HttpResponse, Lớp, Các hàm hành  
 sự, 524  
 HttpResponse, Lớp, Các thuộc tính  
 cốt lõi, 523  
 I/O, 27  
 IComponent, Lớp Component, 133

Icon, thuộc tính lớp StatusBarPanel, 171  
ID, thuộc tính lớp Control, 532  
IDispatchImplAttribute, Lớp Runtime.InteropServices, 398  
IDisposable, giao diện, 133  
IDL (Interface Definition Language), 279  
IDL Enumeration, cấu hình hoá một..., 436  
Idle, Lớp Application, 127  
IFormatter, Formatter, 96  
ILGenerator, 312  
ImageIndex, thuộc tính lớp ToolBarButton, 178  
ImageList, Lớp, Thêm một ImageList vào lúc thiết kế, 183  
ImageList, thuộc tính lớp ToolBar, 177  
ImageSize, thuộc tính lớp ToolBar, 177  
IMessage, 350  
IMessageFilter, 130  
IMessageFilter, Lớp Application, 126  
InAttribute, Lớp Runtime.InteropServices, 398  
Increment, lớp Interlocked, 386  
Index, Lớp MenuItem, 164  
Informational version, version identifier, 237  
Init, tình huống Page, 520  
Initialize, tình huống, 485  
Interface Hierarchy Conversion, 442  
InterfacesDispatch, 441  
InterfacesDual, 441  
InterfacesUnknown, 441  
InterfaceTypeAttribute, Lớp Runtime.InteropServices, 398

Interlocked, Lớp Threading, 371  
Interlocked, Lớp, 386  
Interoperability, 396  
Interoperability, Từ COM qua .NET, 452  
Interrupt(), hàm lớp Thread, 373  
Interrupt(), hàm, 378  
Interval, thuộc tính lớp Timer, 174  
InvokeMember(), hàm lớp Type, 321  
InvokeMember(), hàm lớp Type, 417  
IsAlive, thuộc tính lớp Thread, 373  
IsBackground, thuộc tính lớp Thread, 373  
IsClientConnected, thuộc tính HttpResponseMessage, 523  
ISerializable, 109  
ISerializable, Formatter, 96  
IsMDIChild, 153  
IsMDIContainer, 153  
isolated storage, 106  
IsolatedStorageFileStream, Lớp, 107  
IsParent, Lớp Menu, 157  
IsPostBack, thuộc tính Page, 520  
IsSecureConnection, lớp HttpRequest, 522  
JavaScript, 498  
Join(), hàm lớp Thread, 373, 377  
kernel32.dll, 399  
Kết nối, Thụ lý nhiều..., 82  
Keyboard, Đáp ứng tình huống, 146  
KeyCode, 147  
KeyData, 147  
KeyEventArgs, 147  
KeyEventArgs, Các thuộc tính của.... See  
KeyEventArgs, Lớp, 147  
Keys, thuộc tính  
HttpApplicationState, 525  
Kịch bản phía Client, 497

Kiểm tra hợp lệ đối với trang HTML, 500  
 LastAccessTime, 30  
 LastWriteTime, 30  
 Late binding (gắn kết trễ), 415  
 Late binding, tìm hiểu..., 308  
 LayoutMDI, 154  
 LCE (Loosely Coupled Events)., 470  
 Length, 42, 54  
 Load View State, tình huống, 485  
 Load(), 344  
 Load, tình huống Page, 520  
 Load, tình huống, 485  
 LoadPostData(), hàm, 485  
 LoadViewState(), hàm, 485  
 LocalBuilder, 312  
 Low-Level COM Interfaces, che dấu, 406  
 Mạch trình, 370, 371  
 Mạch trình, Cho treo mạch trình, 378  
 Mạch trình, Đặt tên thân thiện cho mạch trình, 376  
 Mạch trình, Khai tử một mạch trình, 378  
 Mạch trình, Khởi động các mạch trình, 373  
 Mạch trình, Ráp lại các mạch trình, 377  
 MainMenu, Lớp, 159  
 MaintainState, thuộc tính lớp Control, 532  
 Manifest, 231  
 manifest, 232  
 Manifest, Các module trên manifest, 233  
 MarshalByRefObject, 351  
 Marshaling, 342  
 Marshalling, Khai báo phương pháp, 350

Marshalling, Marshalling với proxy, 350  
 Marshalling, Thí dụ marshalling xuyên app domain, 352  
 Marshalling, Xuyên phạm trù, 357  
 Marshalling, Xuyên ranh giới App Domain, 349  
 MaximizeBox, 153  
 MDIChildActivate, 154  
 MdiListItem, Lớp Menu, 157  
 MemoryStream và FileStream, Phối hợp với nhau, 58  
 MemoryStream, Làm việc với ..., 57  
 MemoryStream, Lớp, 28  
 MemoryStream, Lớp, Các thành viên của ..., 58  
 Menu\$MenuItemCollection, Lớp, 158  
 Menu\$MenuItemCollection, Lớp, Các thành viên của lớp..., 158  
 Menu, 153  
 Menu, Lớp, 156  
 Menu, Lớp, Các thành viên, 157  
 MenuItem, Lớp, 164  
 MenuItem, Lớp, Các thuộc tính của..., 164  
 MenuItems, 158  
 MenuItems, Lớp Menu, 157  
 MergedMenu, 153  
 MergeMenu, Lớp Menu, 157  
 MergeOrder, Lớp MenuItem, 165  
 MergeType, Lớp MenuItem, 165  
 MetaInfo, cửa sổ, Metadata, 249  
 MethodBuilder, 312  
 Microsoft.Win32, namespace, 209  
 MinimizeBox, 153  
 MinWidth, thuộc tính lớp  
 StatusBarPanel, 171

Mô phỏng một nguồn lực được chia sẻ sử dụng, 383  
Modifiers, 148  
Module manifest, 234  
ModuleBuilder, 312  
Modules, thành phần kết cấu của assembly, 230  
Monitor, Lớp Threading, 372  
MouseButtons, Enumeration, 145  
MouseEventArgs, Lớp, 144  
MouseEventArgs, Lớp, Các thuộc tính của ..., 144  
MouseUp, Đáp ứng tình huống, 143  
MoveTo(), 31, 42  
MSIL Code, 231  
MSMQ (Microsoft Message Queue), 470  
Mục chọn trình đơn, Cho hiển thị dòng nhắc, 175  
Multi-module assembly, Thử khảo sát, 250  
Multi-module assembly, Xây dựng một..., 252  
Mutex, Lớp Threading, 372  
Name, 30  
Name, Lớp RegistryKey, 212  
Name, thuộc tính lớp Thread, 373, 376  
NET-to-COM, Các vấn đề được đặt ra do..., 463  
Network I/O, 76  
Network Streaming Server, Tạo..., 78  
NetworkStream, 80  
NetworkStream, Lớp, 28  
NewLine, 61  
NonSerialized, attribute, 93  
Nút chuột nào bị click, 145  
Ô control WebForm "bấm sinh", 533

Ô control WebForms bấm sinh, Liệt kê các ..., 534  
object construction string, 470  
Object Graph, 93  
Object handle, 349  
Object Interface, 458  
Object pooling, 470  
ObjectIDGenerator, Lớp, 97  
ObjectManager, Lớp, 97  
OLE/COM Object Viewer, Nhìn xem kiểu dữ liệu, 459  
OleInitialize, 127  
OleRequired(), Lớp Application, 127  
OnPreRender(), hàm, 485  
OnResize(), 154  
OnTick(), Hàm lớp Timer, 174  
Open(), 42  
Open(), Lớp FileInfo, 45  
OpenRead(), 42  
OpenRead(), Lớp FileInfo, 49  
OpenRemoteBaseKey(), Lớp RegistryKey, 212  
OpenSubKey(), Lớp RegistryKey, 212  
OpenText(), Lớp FileInfo, 49  
OpenText, 43  
OpenWrite(), Lớp FileInfo, 49  
OpenWrite, 43  
OutAttribute, Lớp  
Runtime.InteropServices, 398  
Output, thuộc tính HttpResponseMessage, 523  
OutputStream, thuộc tính HttpResponseMessage, 523  
OwnerDraw, Lớp MenuItem, 165  
Page, thuộc tính lớp Control, 532  
Page.Application, thuộc tính, 524  
Page.Request, Làm việc với thuộc tính, 521



Paint, Tình huống, 151  
 PaintEventArgs, 151  
 PaintEventArgs, Các thuộc tính của lớp, 151  
 Panels, 170  
 Panels, collection của StatusBar, 170  
 Parameter Attribute Conversion, 441  
 Params, lớp HttpRequest, 522  
 Parent, 31  
 ParentForm, thuộc tính, Lớp ContainerControl, 152  
 PartialPush, thuộc tính lớp  
 ToolBarButton, 178  
 Peek(), 63  
 PeekChar(), 70  
 Phạm trù (context), 355  
 Phạm trù, Đối tượng context-bound & context-agile, 356  
 Phản chiếu, Liệt kê các thành viên của một lớp, 303  
 Phản chiếu, Phản chiếu trên một kiểu dữ liệu, 303  
 Phiên bản Assembly, Ghi nhận thông tin về..., 274  
 Phiên bản custom, Khai báo cơ chế..., 278  
 Phiên bản, cơ chế ... trong assembly, 267  
 PInvoke, 399  
 PInvoke, Thí dụ sử dụng, 399  
 PInvoke, Thí dụ sử dụng, 403  
 Position, 54  
 POST Submissions, Đáp lại, 506  
 PreFilterMessage, 130  
 PreRender, tình huống, 485  
 PreserveSig, vùng mục tin của DllImportAttribute, 401  
 Priority, thuộc tính lớp Thread, 373  
 Private assembly, 236

Private assembly, Nhận diện một..., 261  
 Private Assembly, Tập tin XML Configuration và..., 262  
 Private assembly, Tìm hiểu về..., 259  
 Probing, cơ bản về..., 261  
 Process Postback Data, tình huống, 485  
 Process, 343  
 ProcessExit, tình huống lớp AppDomain, 345  
 ProcessTabKey(), Hàm, Lớp ContainerControl, 152  
 ProductName, Lớp Application, 127  
 ProductVersion, Lớp Application, 127  
 ProgIdAttribute, Lớp  
 Runtime.InteropServices, 398  
 Project | Add Class, 119  
 Project | Add Reference..., 120  
 PropertyBuilder, 312  
 Proxy, 343, 350  
 Pushed, thuộc tính lớp  
 ToolBarButton, 178  
 Query String, 503  
 Query String, Phân tích ngữ nghĩa, 503  
 QueryString, lớp HttpRequest, 522  
 Queued Component, 470  
 Quick Fix Engineering, 267  
 Race condition, 394  
 Radio Buttons, tạo một group..., 535  
 RadioCheck, Lớp MenuItem, 165  
 RaisePostDataChange Event(), hàm, 485  
 RangeValidator, 544  
 RawUrl, lớp HttpRequest, 522  
 Read(), 54, 63, 70  
 Read(), Lớp FileStream, 56

ReadBlock(), 63  
ReadByte(), 54  
ReadByte(), Lớp FileStream, 56  
ReadLine(), 63  
ReadToEnd(), 63  
ReadXXXX(), 70  
RealProxy, lớp abstract, 350  
Redirect(), hàm HttpResponseMessage, 524  
Reflection Emit, 292  
Reflection emit, 318  
Reflection, namespace, 292  
Reflection, Quan sát nội dung một assembly, 299  
Reflection, Tìm hiểu về reflection, 292  
Reflection, Tìm hiểu về..., 279  
Refresh(), Lớp FileSystemInfo, 31  
regasm.exe, 456  
Regasm.exe.(tắt chữ Register Assembly)., 453  
regedit.exe, 209  
Regedt32.exe, 209  
RegisterChannel, hàm của lớp ChannelServices, 360  
RegisterWellKnownServiceType, hàm, 365  
Registry .NET, Lớp, 211  
Registry, 209  
Registry, Lớp Microsoft.Win32, 209  
Registry, Quan sát các mục vào, 460  
RegistryHive, Lớp Microsoft.Win32, 209  
RegistryKey, Lớp Microsoft.Win32, 209  
RegistryKey, Lớp, 212  
Regsvr32.exe, 425  
RegularExpressionValidator, 544  
Remoting, 342  
Remoting, Tìm hiểu, 358

RemotingConfiguration, Lớp, 360  
Remove, 158  
RemoveMessageFilter(), Lớp Application, 127  
RemoveMessageFilter, Lớp Application, 131  
Render(), hàm, 486  
Request, thuộc tính Page, 520  
Request.QueryString, hàm, 506  
RequestType, lớp HttpRequest, 522  
RequiredFieldValidator, 544  
Resize, tình huống, Lớp Form, 138  
ResourceResolve, tình huống lớp AppDomain, 345  
Resources, 231  
Response, thuộc tính Page, 520  
Resume(), hàm lớp Thread, 373  
Rich Control WebForm, Liệt kê các..., 536  
Run(), hàm, Lớp Application, 126  
Run(), Lớp Application, 127  
Runtime Callable Wrapper (RCW, 404  
Sàn sinh hằng loạt (Serialization), 92  
Sàn Sinh Hằng Loạt, 27  
sản sinh hằng loạt, Cấu hình các đối tượng cho ..., 93  
Sàn sinh hằng loạt, SOAP Formatter, 102  
SaveState, tình huống, 485  
SaveViewState(), hàm, 486  
ScrollableControl, Lớp, 151  
Security boundary, 237  
Seek(), 54, 70  
Send Postback Change Modifications, tình huống, 485  
Serializable, attribute, 93  
serializable, Một lớp Car đơn giản, 95

Serialization và Deserialization một đối tượng, 100  
Serialization, 27  
SerializationBinder, Lớp, 97  
SerializationInfo, 110  
SerializationInfo, Lớp, 97  
Serialize(), Lớp, 98  
Server, thuộc tính Page, 520  
Server, Khai báo một server với một interface, 359  
Server, Xây dựng một server, 359  
ServerVariables, lớp HttpRequest, 522  
Session, thuộc tính Page, 520  
SetAppDomainPolicy(), hàm của lớp AppDomain, 344  
SetCreationTime(), 42  
SetData(), hàm của lớp AppDomain, 344  
SetData(), hàm lớp Thread, 372  
SetLastAccessTime(), 42  
SetLastError, vùng mục tin của DllImportAttribute, 401  
SetLastWriteTime(), 42  
SetLength(), 54  
SetValue(), Lớp RegistryKey, 212  
Shared assembly, Strong name, 265  
Shared Assembly, Sử dụng một..., 272  
Shared Assembly, Thử tìm hiểu..., 265  
Shared assembly, Xây dựng một..., 269  
Shift, 148  
Shortcut, Lớp MenuItem, 165  
Show(), hàm, Lớp MessageBox, 128  
ShowDialog(), 154  
ShowInTaskBar, 153  
ShowPanels, 170

ShowPanels, thuộc tính của StatusBar, 169  
ShowShortcut, Lớp MenuItem, 165  
ShowToolTips, thuộc tính lớp ToolBar, 177  
Sink, 343, 350  
Site, thuộc tính, Lớp Component, 133  
SizingGrip, 170  
SizingGrip, thuộc tính của StatusBar, 170  
Sleep(), hàm lớp Thread, 372, 378  
sn.exe, trình tiện ích, 269  
SOAP Formatter, 102  
SOAP, Nghi thức, 96  
SoapFormatter, Formatter, 96  
Socket, 76  
StackBuilder, 350  
Start(), 77  
Start(), hàm lớp Thread, 373  
Start(), Hàm lớp Timer, 174  
StartPosition, 153  
StartupPath, Lớp Application, 127  
StaticObjects, thuộc tính HttpApplicationState, 525  
StatusBar, Lớp, 169  
StatusBar, ô control, 169  
StatusBarPanel, Lớp, 171  
StatusCode, thuộc tính HttpResponse, 523  
StatusDescription, thuộc tính HttpResponse, 523  
Stop(), Hàm lớp Timer, 174  
Stream Reader, StreamWriter, Lớp, 28  
Stream, Các thành viên lớp abstract, 54  
Stream, Lớp abstract, 28, 53

Streaming Network Client, Tạo một..., 80  
 streaming network client, Thiết đặt một..., 81  
 streaming network server, Thí dụ sử dụng..., 79  
 StreamingContext, 110  
 StreamingContextStates, enumeration, 110  
 StreamingContextStates, enumeration, Các thành viên của..., 111  
 StreamReader, 49  
 StreamReader, Lớp, 61  
 StreamWriter, Lớp, 61  
 String Reader, StringWriter, Lớp, 28  
 StringReader, Lớp, 61  
 StringWriter và StringReader, Làm việc với..., 66  
 StringWriter, Lớp, 61  
 Strong Name, Tìm hiểu, 268  
 Style, thuộc tính lớp StatusBarPanel, 171  
 Style, thuộc tính lớp ToolBarButton, 178  
 SubKeyCount, Lớp RegistryKey, 212  
 SuppressContent, thuộc tính HttpResponseMessage, 523  
 Suspend(), hàm lớp Thread, 373  
 Synchronization, 382  
 System Registry, Các lớp thao tác ..., 209  
 System Registry, Tương tác với..., 208  
 System.  
 Runtime.Serialization.Formatter, Namespace, 96

System.AppDomain, Các thành viên của lớp AppDomain, 344  
 System.Attribute, 280  
 System.Diagnostics, namespace, 217  
 System.dll, 120  
 System.Drawing, 141  
 System.EnterpriseServices, namespace, 470  
 System.IO, namespace, 28  
 System.IO.DirectoryNotFoundException., 32  
 System.MulticastDelegate, 448  
 System.Reflection, Các thành viên của System.Reflection, 298  
 System.Reflection, namespace, 298  
 System.Reflection.Emit, namespace, 311  
 System.Reflection.Emit, namespace, Các thành viên cốt lõi..., 311  
 System.Reflection.Emit, namespace, Tìm hiểu..., 311  
 System.Reflection.MemberInfo, Lớp, Sử dụng thế nào?, 298  
 System.Runtime, 282  
 System.Runtime.Serialization, 109  
 System.Runtime.InteropServices, namespace, Các lớp cốt lõi, 398  
 System.Runtime.Serialization, Các lớp của namespace..., 97  
 System.Runtime.Serialization, Namespace, 97  
 System.Text.Reference, 61  
 System.Threading, namespace, 370, 371  
 System.Threading, namespace, Các thành viên cốt lõi, 371  
 System.Threading.Interlocked, 386  
 System.Threading.Monitor, 388  
 System.Type, Lớp, 292

System.Type, Lớp, Khảo sát, 292  
 System.TypedReference, Lớp, 292  
 System.Web, Các lớp cốt lõi, 510  
 System.Web, namespace, 482  
 System.Web.UI, namespace, 482  
 System.Web.UI.HtmlControls, namespace, 494  
 System.Web.UI.Page, lớp, Các thuộc tính cốt lõi, 519  
 System.Windows.Forms, Các lớp cốt lõi, 118  
 System.Windows.Forms, Namespace, 116  
 System.Windows.Forms.dll, 120  
 Tái dụng đoạn mã, Assembly khuyến khích..., 236  
 Tài liệu HTML, Cấu trúc cơ bản, 486  
 Tập tin & Thư mục, 28  
 Tập tin cấu hình ứng dụng, 262  
 Tập tin PE (Portable Executable), 230  
 Tập tin văn bản, Làm việc với ..., 60  
 Tập tin văn bản, Viết và Đọc, 62  
 Tập tin, Thay đổi các tập tin, 50  
 TCP/IP, Nghi thức, 76  
 TcpClient, 80  
 TcpListene, 78  
 TcpListener, Lớp, 78  
 tcpSocket, 84  
 Text, Lớp MenuItem, 165  
 Text, thuộc tính lớp ToolBarButton, 179  
 TextBox, tạo một ..., 536  
 TextReader, Lớp, Các thành viên của ..., 63  
 TextReader, TextWriter, Lớp, 28  
 TextWriter, Lớp, Các thành viên của ..., 61  
 Thanh công cụ, tạo một..., 176

Thanh công cụ, tạo vào lúc thiết kế, 182  
 Thanh công cụ. Đồng bộ hoá, 185  
 Thanh tình trạng, Đồng bộ hoá với trình đơn, 173  
 Thanh tình trạng, tạo một..., 171  
 Thanh tình trạng, Tìm hiểu, 169  
 Thread, 371  
 Thread, Lớp Threading, 372  
 Thread, Lớp, Các thành viên cấp đối tượng, 373  
 Thread, Lớp, Các thành viên của lớp ..., 372  
 ThreadExit, Lớp Application, 127  
 ThreadInterruptedException, 379  
 ThreadPool, Lớp Threading, 372  
 ThreadStart, Lớp delegate, 373  
 ThreadStart, Lớp Threading, 372  
 ThreadState, thuộc tính lớp Thread, 373  
 Thụ lý nhiều kết nối, 82  
 Thư Mục & Tập tin, 28  
 Thư mục ảo, 480  
 Thư mục con, Rảo xem, 35  
 Thư mục con, tạo sử dụng lớp DirectoryInfo, 38  
 Tick, Tình huống lớp Timer, 174  
 Timer, Lớp Threading, 372  
 Timer, Lớp, Các thành viên, 174  
 TimerCallBack, Lớp Threading, 372  
 Tình huống COM, Chặn hứng các..., 448  
 Tình huống postback, 484  
 Tình huống, Điều khiển các..., 140  
 Tình trạng đối tượng (state object), 72  
 TlbExp.exe (tắt chữ Type Library Export), 453  
 tlbimp.exe, 404, 412

Tô vẽ, Căn bản về..., 151  
 ToArray(), 58  
 ToLongTimeString(), 172  
 ToolBar, Lớp, 176  
 ToolBar, Lớp, Các th, 177  
 ToolBar, ô control, 176  
 ToolBarButton, Lớp, 178  
 ToolBarButton. Lớp, 178  
 ToolBarButtonClickEventArgs, 180  
 ToolBarButtonClickEventHandler, 180  
 Tools | OLE/COM Object Viewer, 410  
 ToolTipText, thuộc tính lớp  
 StatusBarPanel, 171  
 ToolTipText, thuộc tính lớp  
 ToolBarButton, 179  
 Trang HTML, Định dạng văn bản, 488  
 Transparent proxy, 350  
 Trình đơn shortcut. Tạo một..., 162  
 Trình đơn, Tạo một hệ thống ..., 159  
 Trình đơn, Tạo trình đơn với  
 Windows Form, 156  
 Tương tác với .DLL viết theo C, 399  
 Type boundary, 237  
 Type Library Conversion, 440  
 Type Library, kết sinh, 456  
 Type Library, nhập khẩu, 412  
 Type Metadata, 231  
 Type, Lớp, 293  
 Type, Lớp, Các thành viên lớp Type, 293  
 Type, Lớp, Nhận về một đối tượng Type, 293  
 Type, Lớp, Sử dụng lớp Type, 294  
 TypeBuilder, 312  
 TypeLibTypeAttribute, 441  
 UDP, Nghi thức, 76

Ứng dụng ASP .NET Web, Gỡ rối & Lăn theo dấu vết, 525  
 Ứng dụng ASP .NET, xây dựng một, 508  
 Ứng dụng ASP.NET Web, thử xem kiến trúc của..., 518  
 Ứng dụng Web, 479  
 Ứng dụng Windows, Tạo một ..., 116  
 Unload(), hàm của lớp AppDomain, 344  
 Unload, tình huống Page, 520  
 Unmanaged Code, 396  
 User Interface, xây dựng, 494  
 user32.dll, 399  
 UserControl, Lớp, 117  
 UserHostAddress, lớp HttpRequest, 522  
 UserHostName, lớp HttpRequest, 522  
 Validation Controls, 544  
 ValidationSummary, 544  
 ValueCount, Lớp RegistryKey, 212  
 Version identifier, của một assembly, 237  
 Version, cơ chế, 229  
 View | Toolbars | Formatting, thanh công cụ, 492  
 ViewState, thuộc tính, 485  
 Virtual Directories, Tìm hiểu, 480  
 Visible, thuộc tính lớp Control, 532  
 Visible, thuộc tính lớp  
 ToolBarButton, 178, 179  
 Visual Studio .NET HTML Editors, 491  
 Wait(), hàm lớp Monitor, 389  
 WaitCallback, Lớp Threading, 372  
 WaitHandle, Lớp Threading, 372  
 Web Form, Chu kỳ sống của ..., 484  
 Web Form, Một ứng dụng..., 548

Web Forms, Các tình huống trên, 483  
Web Forms, tìm hiểu về, 482  
Web server, 479  
Web.config, khảo sát tập tin, 516  
WebControl, Lớp cơ bản, Các thuộc tính, 533  
WebForm Controls, Các loại ô control, 533  
WebForm Controls, Dẫn xuất các ô..., 532  
WebForm Controls, Làm việc với các ô..., 530  
WebForm Controls, Tìm hiểu, 528  
WebForm Data Control, 541  
WebForm Validation Controls, 544  
WebForms Control, Thụ lý tình huống, 546  
Well-known, đối tượng, 358  
WellKnownObjectMode, 361

Windows Form, Tạo một dự án sử dụng VS IDE, 122  
Windows Form, Tạo một Windows Form đơn giản, 119  
Windows Forms, Tương tác với các lớp..., 119  
Windows.Forms, Tổng quan, 117  
WindowState, 153  
Wrappable, thuộc tính lớp ToolBar, 177  
Write(), 54, 61, 70  
Write(), hàm HttpResponseMessage, 524  
Write(), Lớp FileStream, 56  
WriteByte(), 54  
WriteByte(), Lớp FileStream, 56  
WriteFile(), hàm HttpResponseMessage, 524  
WriteLine(), 61  
WriteTo(), 58  
Xuất nhập dữ liệu bất đồng bộ, 72  
Xuất nhập dữ liệu trên mạng, 76  
Xuất Nhập Dữ Liệu, 27

Chịu trách nhiệm xuất bản:

TRẦN ĐÌNH VIỆT

Biên tập:

TRUNG HIẾU

Sửa bản in:

LÊ DUNG

Bìa:

HOÀNG NGỌC GIAO

NHÀ XUẤT BẢN TỔNG HỢP TP. HCM

62 Nguyễn Thị Minh Khai – Q.1

ĐT: 82225340 – 8296764 – 8220405 – 8296713 – 8223637

Fax: 8222726 – Email: [nxbtpHCM@bdvn.vnd.net](mailto:nxbtpHCM@bdvn.vnd.net)

In 1000 cuốn, khổ 16 x 22cm, tại Xí nghiệp in FAHASA.  
Giấy phép xuất bản số 910-9/XB-QLXB ký ngày 1-7-2004.  
In xong và nộp lưu chiểu tháng 4-2005



