APPENDIX B

Platform-Independent .NET Development with Mono

This appendix introduces you to the topic of cross-platform C# and .NET development using an open source implementation of .NET named *Mono* (in case you are wondering about the name, *mono* is a Spanish word for monkey, which is a reference to the various monkey-mascots used by the initial creators of the Mono platform, Ximian Corporation). In this appendix, you will learn about the role of the Common Language Infrastructure (CLI), the overall scope of Mono, and various Mono development tools. At the conclusion of this appendix-and given your work over the course of this book—you will be in a perfect position to dig further into Mono development as you see fit.

■ **Note** If you require a detailed explanation of cross-platform .NET development, I recommend picking up a copy of *Cross-Platform .NET Development: Using Mono, Portable .NET, and Microsoft .NET* by Mark Easton and Jason King (Apress, 2004).

The Platform-Independent Nature of .NET

Historically speaking, when programmers used a Microsoft development language (e.g., VB6) or Microsoft programming framework (e.g., MFC, COM, or ATL), they had to resign themselves to building software that (by-and-large) executed only on the Windows family of operating systems. Many .NET developers, accustomed to previous Microsoft development options, are frequently surprised when they learn that .NET is *platform-independent*. But it's true. You can create, compile, and execute .NET assemblies on operating systems other than Microsoft Windows.

Using open source .NET implementations such as Mono, your .NET applications can find happy homes on numerous operating systems, including Mac OS X, Solaris, AIX, and numerous flavors of Unix/Linux. Mono also provides an installation package for (surprise, surprise) Microsoft Windows. Thus, it is possible to build and run .NET assemblies on the Windows operating system, without ever installing the Microsoft .NET Framework 4.0 SDK or the Visual Studio 2010 IDE.

■ **Note** Be aware that the Microsoft .NET Framework 4.0 SDK and Visual Studio 2010 are your best options for building .NET software for the Windows family of operating systems.

Even after developers learn about .NET code's cross-platform capabilities, they often assume that the scope of platform-independent .NET development is limited to little more than *Hello World* console applications. The reality is that you can build production-ready assemblies that use ADO.NET, Windows Forms (in addition to alternative GUI toolkits such as GTK# and Cocoa#), ASP.NET, LINQ, and XML web services by taking advantage of many of the core namespaces and language features you have seen featured throughout this book.

The Role of the CLI

.NET's cross-platform capabilities are implemented differently than the approach taken by Sun Microsystems and its handling of the Java programming platform. Unlike Java, Microsoft itself does not provide .NET installers for Mac, Linux, and so on. Rather, Microsoft has released a set of formalized specifications that other entities can use as a road map for building .NET distributions for their platform(s) of choice. Collectively, these specifications are termed the *Common Language Infrastructure* (CLI).

As briefly mentioned in Chapter 1, Microsoft submitted two formal specifications to ECMA (European Computer Manufacturers Association) when it released C# and the .NET platform to the world at large. Once approved, Microsoft submitted these same specifications to the International Organization for Standardization (ISO), and these specifications were ratified shortly thereafter.

So, why should you care? These two specifications provide a road map for other companies, developers, universities, and other organizations to build their own custom distributions of the C# programming language and the .NET platform. The two specifications in question are:

- *ECMA-334*: Defines the syntax and semantics of the C# programming language.
- *ECMA-335*: Defines many details of the .NET platform, collectively called the *Common Language Infrastructure*.

ECMA-334 tackles the lexical grammar of C# in an extremely rigorous and scientific manner (as you might guess, this level of detail is quite important to those who want to implement a C# compiler). However, ECMA-335 is the meatier of the two specifications, so much so that it has been broken down into six partitions (see Table B-1).

Table B-1. *ECMA-335 Specification Partitions*

ECMA-335 Partition	Meaning in Life
Partition I: Concepts and Architecture	Describes the overall architecture of the CLI, including the rules of the Common Type System, the Common Language Specification, and the mechanics of the .NET runtime engine.
Partition II: Metadata Definition and Semantics	Describes the details of the .NET metadata format.
Partition III: CIL Instruction Set	Describes the syntax and semantics of the common intermediate language (CIL) programming language.
Partition IV: Profiles and Libraries	Provides a high-level overview of the minimal and complete class libraries that must be supported by a CLI-compatible .NET distribution.
Partition V: Debug Interchange Formats	Provides details of the portable debug interchange format (CILDB). Portable CILDB files provide a standard way to exchange debugging information between CLI producers and consumers.
Partition VI: Annexes	Represents a collection of *odds and ends*; it clarifies topics such as class library design guidelines and the implementation details of a CIL compiler.

The point of this appendix is not to dive into the details of the ECMA-334 and ECMA-335 specifications—nor must you know the ins-and-outs of these documents to understand how to build platform-independent .NET assemblies. However, if you are interested, you can download both of these specifications for free from the ECMA website (`http://www.ecma-international.org/publications/standards`).

The Mainstream CLI Distributions

To date, you can find two mainstream implementations of the CLI beyond Microsoft's CLR, Microsoft Silverlight, and the Microsoft NET Compact Framework (see Table B-2).

Table B-2. *Mainstream .NET CLI Distributions*

CLI Distribution	Supporting Website	Meaning in Life
Mono	`www.mono-project.com`	Mono is an open source and commercially supported distribution of .NET sponsored by Novell Corporation. Mono is targeted to run on many popular flavors of Unix/Linux, Mac OS X, Solaris, and Windows.
Portable .NET	`www.dotgnu.org`	Portable .NET is distributed under the GNU General Public License. As the name implies, Portable .NET intends to function on as many operation systems and architectures as possible, including esoteric platforms such as BeOS, Microsoft Xbox, and Sony PlayStation (no, I'm not kidding about the last two!).

Each of the CLI implementations shown in Table B-2 provide a fully function C# compiler, numerous command-line development tools, a global assembly cache (GAC) implementation, sample code, useful documentation, and dozens of assemblies that constitute the base class libraries.

Beyond implementing the core libraries defined by Partition IV of ECMA-335, Mono and Portable .NET provide Microsoft-compatible implementations of `mscorlib.dll`, `System.Core.dll`, `System.Data.dll`, `System.Web.dll`, `System.Drawing.dll`, and `System.Windows.Forms.dll` (among many others).

The Mono and Portable .NET distributions also ship with a handful of assemblies specifically targeted at Unix/Linux and Mac OS X operating systems. For example, Cocoa# is a .NET wrapper around the preferred Mac OX GUI toolkit, Cocoa. In this appendix, I will not dig into these OS-specific binaries; instead, I'll focus on using the OS-agonistic programming stacks.

■ **Note** This appendix will not examine Portable .NET; however, it is important to know that Mono is not the only platform-independent distribution of the .NET platform available today. I recommend you take some time to play around with Portable .NET in addition to the Mono platform.

The Scope of Mono

Given that Mono is an API built on existing ECMA specifications that originated from Microsoft, you would be correct in assuming that Mono is playing a constant game of catch up as newer versions of Microsoft's .NET platform are released. At the time of this writing, Mono is compatible with C# 2008 (work on the new C# 2010 language features are in development as I type) and .NET 2.0. This means you can build ASP.NET websites, Windows Forms applications, database-centric applications using ADO.NET, and (of course) simple console applications.

Currently, Mono is *not* completely compatible with the new features of C# 2010 (e.g., optional arguments and the `dynamic` keyword) or with all aspects of .NET 3.0-4.0. This means that Mono applications are currently unable (again, at the time of this writing) to use the following Microsoft .NET APIs:

- Windows Presentation Foundation (WPF)
- Windows Communication Foundation (WCF)
- Windows Workflow Foundation (WF)
- LINQ to Entities (however, LINQ to Objects and LINQ to XML are supported)
- Any C# 2010 specific language features

Some of these APIs might eventually become part of the standard Mono distribution stacks. For example, according to the Mono website, future versions of the platform (2.8-3.0) will provide support for C# 2010 language features and .NET 3.5-4.0 platform features.

In addition to keeping-step with Microsoft's core .NET APIs and C# language features, Mono also provides an open source distribution of the Silverlight API named *Moonlight*. This enables browsers that run under Linux based operating systems to host and use Silverlight / Moonlight web applications. As you might already know, Microsoft's Silverlight already includes support for Mac OS X; and given the Moonlight API, this technology has truly become cross-platform.

Mono also supports some .NET based technologies that do *not* have a direct Microsoft equivalent. For example, Mono ships with GTK#, a .NET wrapper around a popular Linux-centric GUI framework named GTK. One compelling Mono-centric API is MonoTouch, which allows you to build applications for Apple iPhone and iTouch devices using the C# programming language.

■ **Note** The Mono website includes a page that describes the overall road map of Mono's functionality and the project's plans for future releases (`www.mono-project.com/plans`).

A final point of interest regarding the Mono feature set: Much like Microsoft's .NET Framework 4.0 SDK, the Mono SDK supports several .NET programming languages. While this appendix focuses on C#, Mono also provides support for a Visual Basic compiler, as well as support for many other .NET-aware programming languages.

Obtaining and Installing Mono

Now that you have a better idea what you can do with the Mono platform, let's turn our attention to obtaining and installing Mono itself. Navigate to the Mono website (`www.mono-project.com`) and locate the Download tab to navigate to the downloads page. Here you can download a variety of installers (see Figure B-1).

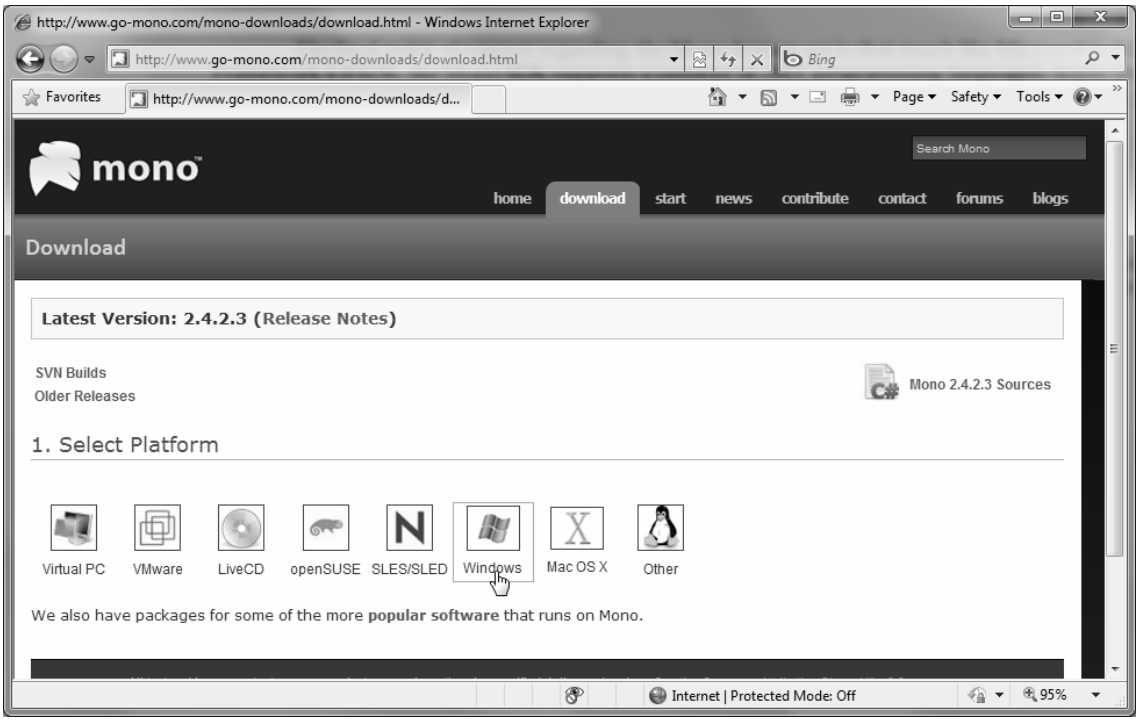

Figure B-1. *The Mono Download Page*

This appendix assumes that you are installing the Windows distribution of Mono (note that installing Mono will not interfere whatsoever with any existing installation of Microsoft .NET or Visual Studio IDEs). Begin by downloading the current, stable Mono installation package for Microsoft Windows and saving the setup program to your local hard drive.

When you run the setup program, you are given a chance to install a variety of Mono development tools beyond the expected base class libraries and the C# programming tools. Specifically, the installer will ask you whether you wish to include GTK# (the open source .NET GUI API based on the Linux-centric GTK toolkit) and XSP (a stand-alone web server, similar to Microsoft's ASP.NET development web server [`webdev.webserver.exe`]). This appendix also assumes you have opted for a Full Installation, which selects each option in the setup script (see Figure B-2).

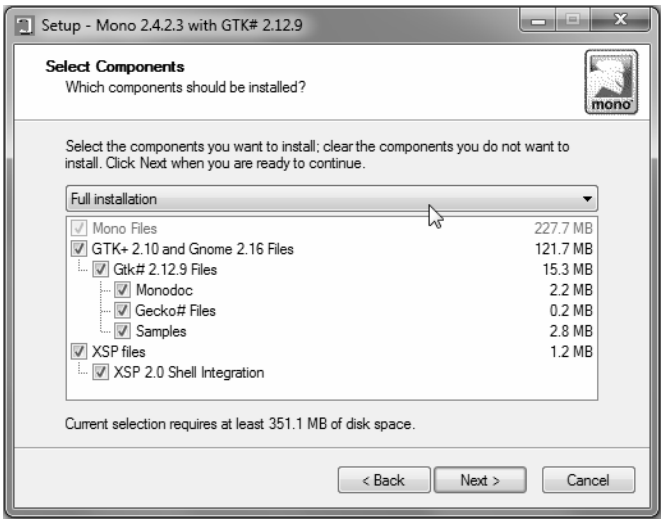

Figure B-2. *Select All Options for Your Mono Installation*

Examining Mono's Directory Structure

By default, Mono installs under `C:\Program Files\Mono-<version>` (at the time of this writing, the latest version of Mono is 2.4.2.3; however, your version number will almost certainly be different). Beneath that root, you can find several subdirectories (see Figure B-3).

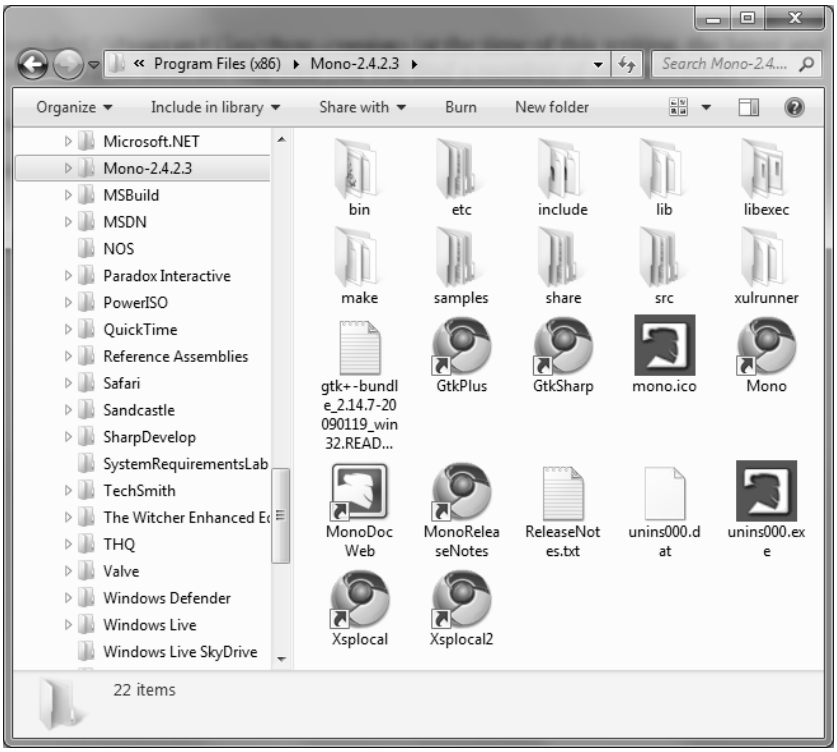

Figure B-3. *The Mono Directory Structure*

For this appendix, you need to concern yourself only with the following subdirectories:

- *bin*: Contains a majority of the Mono development tools, including the C# command-line compilers.
- lib*mono**gac*: Points to the location of Mono's global assembly cache.

Given that you run the vast majority of Mono's development tools from the command line, you will want to use the Mono command prompt, which automatically recognizes each of the command-line development tools. You can activate the command prompt (which is functionally equivalent to the Visual Studio 2010 command prompt) by selecting Start ➤ All Programs ➤ Mono <version> For Windows menu option. To test your installation, enter the following command and press the Enter key:

```
mono --version
```

If all is well, you should see various details regarding the Mono runtime environment. Here is the output on my Mono development machine:

```
Mono JIT compiler version 2.4.2.3 (tarball)
Copyright (C) 2002-2008 Novell, Inc and Contributors. www.mono-project.com
        TLS:           normal
        GC:            Included Boehm (with typed GC)
        SIGSEGV:       normal
        Notification:  Thread + polling
        Architecture:  x86
        Disabled:      none
```

The Mono Development Languages

Similar to the Microsoft's CLR distribution, Mono ships with a number of managed compilers:

- `mcs`: The Mono C# compiler
- `vbnc`: The Mono Visual Basic .NET compiler
- `booc`: The Mono Boo language compiler
- `ilasm`: The Mono CIL compilers

While this appendix focuses only on the C# compilers, you should keep in mind that the Mono project does ship with a Visual Basic compiler. While this tool is currently under development, the intended goal is to bring the world of human-readable keywords (e.g., `Inherits`, `MustOverride`, and `Implements`) to the world of Unix/Linux and Mac OS X (see `www.mono-project.com/Visual_Basic` for more details).

Boo is an object-oriented, statically typed programming language for the CLI that sports a Python-based syntax. Check out `http://boo.codehaus.org` for more details on the Boo programming language. Finally, as you might have guessed, `ilasm` is the Mono CIL compiler.

Working with the C# Compiler

The C# compiler for the Mono project was `mcs`, and it's fully compatible with C# 2008. Like the Microsoft C# command-line compiler (`csc.exe`), `mcs` supports response files, a `/target:` flag (to define the assembly type), an `/out:` flag (to define the name of the compiled assembly), and a `/reference:` flag (to update the manifest of the current assembly with external dependencies). You can view all the options of `mcs` using the following command:

```
mcs -?
```

Building Mono Applications using MonoDevelop

When you install Mono, you will not be provided with a graphical IDE. However, this does not mean you must build all of your Mono applications at the command prompt! In addition to the core framework, you can also download the free MonoDevelop IDE. As its name suggests, MonoDevelop was built using the core code base of SharpDevelop (see Chapter 2).

You can download the MonoDevelop IDE from the Mono website, and it supports installation packages for Mac OS X, various Linux distributions, and (surprise!) Microsoft Windows! Once installed, you might be pleased to find an integrated debugger, IntelliSense capabilities, numerous project templates (e.g., ASP.NET and Moonlight). You can get a taste of what the MonoDevelop IDE brings to the table by perusing Figure B-4.

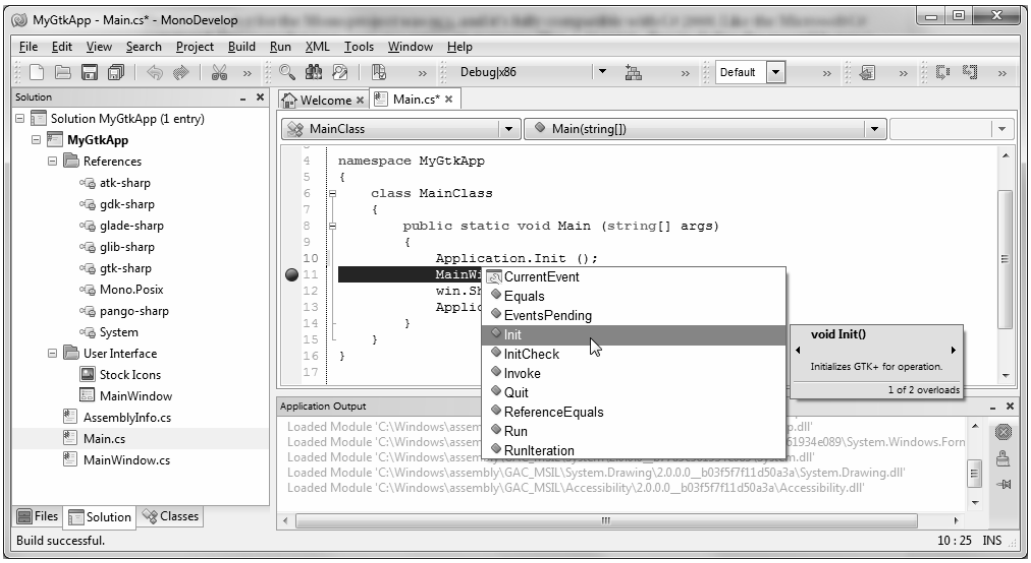

Figure B-4. *The MonoDevelop IDE*

Microsoft-Compatible Mono Development Tools

In addition to the managed compilers, Mono ships with various development tools that are functionally equivalent to tools found in the Microsoft .NET SDK (some of which are identically named). Table B-3 enumerates the mappings between some of the commonly used Mono/Microsoft .NET utilities.

Table B-3. *Mono Command-Line Tools and Their Microsoft .NET Counterparts*

Mono Utility	Microsoft .NET Utility	Meaning in Life
`al`	`al.exe`	Manipulates assembly manifests and builds multifile assemblies (among other things).
`gacutil`	`gacutil.exe`	Interacts with the GAC.
`mono` when run with the `-aot` option as a startup parameter to the executing assembly	`ngen.exe`	Performs a precompilation of an assembly's CIL code.

Table B-3. *Mono Command-Line Tools and Their Microsoft .NET Counterparts* (*continued*)

Mono Utility	Microsoft .NET Utility	Meaning in Life
`wsdl`	`wsdl.exe`	Generates client-side proxy code for XML web services.
`disco`	`disco.exe`	Discovers the URLs of XML web services located on a web server.
`xsd`	`xsd.exe`	Generates type definitions from an XSD schema file.
`sn`	`sn.exe`	Generates key data for a strongly named assembly.
`monodis`	`ildasm.exe`	The CIL disassembler
`ilasm`	`ilasm.exe`	The CIL assembler
`xsp2`	`webdev.webserver.exe`	A testing and development ASP.NET web server

Mono-Specific Development Tools

Mono also ships with development tools for which no direct Microsoft .NET Framework 3.5 SDK equivalents exist (see Table B-4).

Table B-4. *Mono Tools That Have No Direct Microsoft .NET SDK Equivalent*

Mono-Specific Development Tool	Meaning in Life
`monop`	The `monop` (mono print) utility displays the definition of a specified type using C# syntax (see the next section for a quick example).
`SQL#`	The Mono Project ships with a graphical front end (`SQL#`) that allows you to interact with relational databases using a variety of ADO.NET data providers.
`Glade 3`	This tool is a visual development IDE for building GTK# graphical applications.

■ **Note** You can load SQL# and Glade by using the Windows Start button and navigating to the Applications folder within the Mono installation. Be sure to do this because this illustrates definitively how rich the Mono platform has become.

Using monop

You can use the monop utility (short for mono print) to display the C# definition of a given type within a specified assembly. As you might suspect, this tool can be quite helpful when you wish to view a method signature quickly, rather than digging through the formal documentation. As a quick test, try entering the following command at a Mono command prompt:

```
monop System.Object
```

You should see the definition of your good friend, `System.Object`:

```
[Serializable]
public class Object {

        public Object ();

        public static bool Equals (object objA, object objB);
        public static bool ReferenceEquals (object objA, object objB);
        public virtual bool Equals (object obj);
        protected override void Finalize ();
        public virtual int GetHashCode ();
        public Type GetType ();
        protected object MemberwiseClone ();
        public virtual string ToString ();
}
```

Building .NET Applications with Mono

Now let's look at Mono in action. You begin by building a code library named `CoreLibDumper.dll`. This assembly contains a single class type named `CoreLibDumper` that supports a static method named `DumpTypeToFile()`. The method takes a string parameter that represents the fully qualified name of any type within `mscorlib.dll` and obtains the related type information through the reflection API (see Chapter 15), dumping the class member signatures to a local file on the hard drive.

Building a Mono Code Library

Create a new folder on your C: drive named `MonoCode`. Within this new folder, create a subfolder named `CorLibDumper` that contains the following C# file (named `CorLibDumper.cs`):

```
// CorLibDumper.cs
using System;
using System.Reflection;
using System.IO;

// Define assembly version.
[assembly:AssemblyVersion("1.0.0.0")]

namespace CorLibDumper
{
  public class TypeDumper
  {
    public static bool DumpTypeToFile(string typeToDisplay)
    {
      // Attempt to load type into memory.
      Type theType = null;
      try
      {
         // Second parameter to GetType() controls if an
         // exception should be thrown if the type is not found.
         theType = Type.GetType(typeToDisplay, true);
      } catch { return false; }

      // Create local *.txt file.
      using(StreamWriter sw =
         File.CreateText(string.Format("{0}.txt", theType.FullName)))
      {
        // Now dump type to file.
        sw.WriteLine("Type Name: {0}", theType.FullName);
        sw.WriteLine("Members:");
        foreach(MemberInfo mi in theType.GetMembers())
          sw.WriteLine("\t-> {0}", mi.ToString());

      }
      return true;
    }
  }
}
```

Like the Microsoft C# compiler, Mono's C# compilers support the use of response files (see Chapter 2). While you could compile this file by specifying each required argument manually at the command line, you can instead create a new file named `LibraryBuild.rsp` (in the same location as `CorLibDumper.cs`) that contains the following command set:

```
/target:library
/out:CorLibDumper.dll
CorLibDumper.cs
```

You can now compile your library at the command line, as follows:

```
mcs @LibraryBuild.rsp
```

This approach is functionally equivalent to the following (more verbose) command set:

```
mcs /target:library /out:CorLibDumper.dll CorLibDumper.cs
```

Assigning CoreLibDumper.dll a Strong Name

Mono supports the notion of deploying strongly named assemblies (see Chapter 15) to the Mono GAC. To generate the necessary public/private key data, Mono provides the `sn` command-line utility, which functions more or less identically to Microsoft's tool of the same name. For example, the following command generates a new `*.snk` file (specify the `-?` option to view all possible commands):

```
sn -k myTestKeyPair.snk
```

You can tell the C# compiler to use this key data to assign a strong name to `CorLibDumper.dll` by updating your `LibraryBuild.rsp` file with the following additional command:

```
/target:library
/out:CorLibDumper.dll
/keyfile:myTestKeyPair.snk
CorLibDumper.cs
```

Now recompile your assembly:

```
mcs @LibraryBuild.rsp
```

Viewing the Updated Manifest with monodis

Before you deploy the assembly to the Mono GAC, you should familiarize yourself with the `monodis` command-line tool, which is the functional equivalent of Microsoft's `ildasm.exe` (without the GUI front end). Using `monodis`, you can view the CIL code, manifest, and type metadata for a specified assembly. In this case, you want to view the core details of your (now strongly named) assembly using the `--assembly` flag. Figure B-5 shows the result of the following command set:

```
monodis --assembly CorLibDumper.dll
```

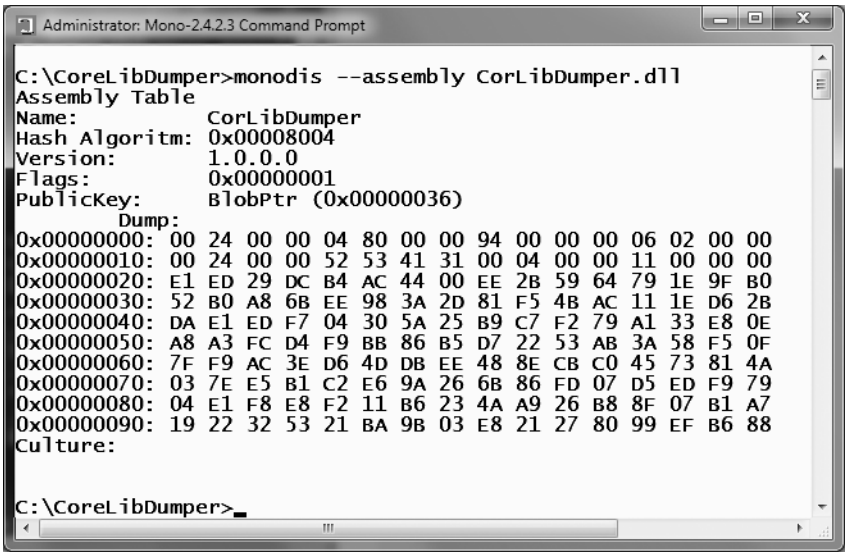

Figure B-5. *Viewing the CIL Code, Metadata, and Manifest of an Assembly with monodis*

The assembly's manifest now exposes the public key value defined in `myTestKeyPair.snk`.

Installing Assemblies into the Mono GAC

So far you have provided `CorLibDumper.dll` with a strong name; you can install it into the Mono GAC using `gacutil`. Like Microsoft's tool of the same name, Mono's `gacutil` supports options to install, uninstall, and list the current assemblies installed under `C:\Program Files\Mono-<version>\lib\mono\gac`. The following command deploys `CorLibDumper.dll` to the GAC and sets it up as a shared assembly on the machine:

```
gacutil -i CorLibDumper.dll
```

■ **Note** Be sure to use a Mono command prompt to install this binary to the Mono GAC! If you use the Microsoft `gacutil.exe` program, you'll install `CorLibDumper.dll` into the Microsoft GAC!

After you run the command, opening the `\gac` directory should reveal a new folder named `CorLibDumper` (see Figure B-6). This folder defines a subdirectory that follows the same naming conventions as Microsoft's GAC (`versionOfAssembly__publicKeyToken`).

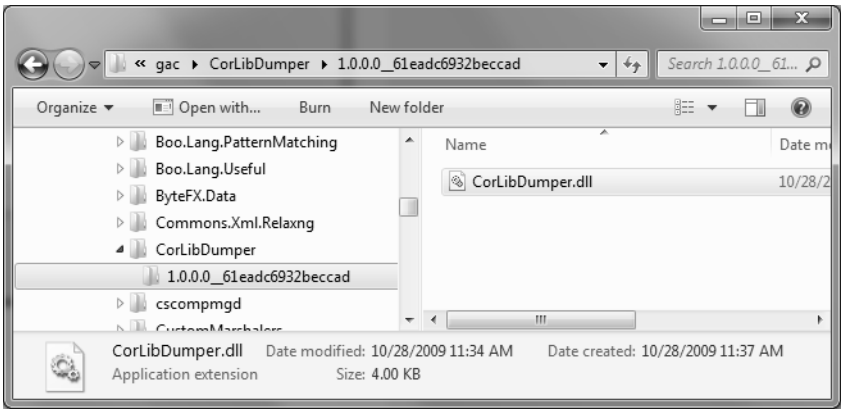

Figure B-6. *Deploying Your Code Library to the Mono GAC*

■ **Note** Supplying the `-l` option to `gacutil` lists out each assembly in the Mono GAC.

Building a Console Application in Mono

Your first Mono client will be a simple, console-based application named `ConsoleClientApp.exe`. Create a new file in your `C:\MonoCode\CorLibDumper` folder, `ConsoleClientApp.cs`, that contains the following `Program` class definition:

```
// This client app makes use of the CorLibDumper.dll
// to dump type information to a file.
using System;
using CorLibDumper;

namespace ConsoleClientApp
{
  public class Program
  {
    public static void Main(string[] args)
    {
      Console.WriteLine("***** The Type Dumper App *****\n");

      // Ask user for name of type.
      string typeName = "";
      Console.Write("Please enter type name: ");
      typeName = Console.ReadLine();

      // Now send it to the helper library.
      if(TypeDumper.DumpTypeToFile(typeName))
```

```
        Console.WriteLine("Data saved into {0}.txt", typeName);
      else
        Console.WriteLine("Error!  Can't find that type...");
    }
  }
}
```

Notice that the `Main()` method prompts the user for a fully qualified type name. The `TypeDumper.DumpTypeToFile()` method uses the user-entered name to dump the type's members to a local file. Next, create a `ClientBuild.rsp` file for this client application and reference `CorLibDumper.dll`:

```
/target:exe
/out:ConsoleClientApp.exe
/reference:CorLibDumper.dll
ConsoleClientApp.cs
```

Now, use a Mono command prompt to compile the executable using `mcs`, as shown here:

```
mcs @ClientBuild.rsp
```

Loading Your Client Application in the Mono Runtime

At this point, you can load `ConsoleClientApp.exe` into the Mono runtime engine by specifying the name of the executable (with the `*.exe` file extension) as an argument to `Mono`:

```
mono ConsoleClientApp.exe
```

As a test, enter `System.Threading.Thread` at the prompt and press the Enter key. You will now find a new file named `System.Threading.Thread.txt` containing the type's metadata definition (see Figure B-7).

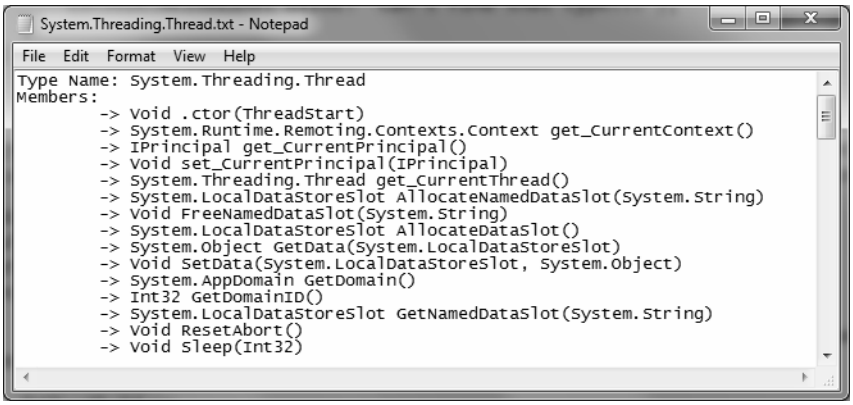

Figure B-7. *The Results of Running Your Client Application*

Before moving on to a Windows Forms–based client, try the following experiment. Use Windows Explorer to rename the `CorLibDumper.dll` assembly from the folder containing the client application to `DontUseCorLibDumper.dll`. You should still be able to run the client application successfully because the only reason you needed access to this assembly when building the client was to update the client manifest. At runtime, the Mono runtime will load the version of `CorLibDumper.dll` you deployed to the Mono GAC.

However, if you open Windows Explorer and attempt to run your client application by double-clicking `ConsoleClientApp.exe`, you might be surprised to find a `FileNotFoundException` is thrown. At first glance, you might assume this is due to the fact that you renamed `CorLibDumper.dll` from the location of the client application. However, the true reason for the error is that you just loaded `ConsoleClientApp.exe` into the Microsoft CLR!

To run an application under Mono, you must pass it into the Mono runtime through `Mono`. If you do not, you will be loading your assembly into the Microsoft CLR, which assumes all shared assemblies are installed into the Microsoft GAC located in the `<%windir%>\Assembly` directory.

■ **Note** If you double-click your executable using Windows Explorer, you will load your assembly into the Microsoft CLR, not the Mono runtime!

Building a Windows Forms Client Program

Before continuing, be sure to rename `DontUseCorLibDumper.dll` back to `CorLibDumper.dll`, so you can compile the next example. Next, create a new C# file named `WinFormsClientApp.cs` and save it in the same location as your current project files. This file defines two class types:

```
using System;
using System.Windows.Forms;
using CorLibDumper;
using System.Drawing;

namespace WinFormsClientApp
{
  // Application object.
  public static class Program
  {
    public static void Main(string[] args)
    {
      Application.Run(new MainWindow());
    }
  }

  // Our simple Window.
  public class MainWindow : Form
  {
    private Button btnDumpToFile = new Button();
    private TextBox txtTypeName = new TextBox();
```

```
    public MainWindow()
    {
      // Config the UI.
      ConfigControls();
    }

    private void ConfigControls()
    {
      // Configure the Form.
      Text = "My Mono Win Forms App!";
      ClientSize = new System.Drawing.Size(366, 90);
      StartPosition = FormStartPosition.CenterScreen;
      AcceptButton = btnDumpToFile;

      // Configure the Button.
      btnDumpToFile.Text = "Dump";
      btnDumpToFile.Location = new System.Drawing.Point(13, 40);

      // Handle click event anonymously.
      btnDumpToFile.Click += delegate
      {
        if(TypeDumper.DumpTypeToFile(txtTypeName.Text))
          MessageBox.Show(string.Format(
            "Data saved into {0}.txt",
            txtTypeName.Text));
        else
          MessageBox.Show("Error!  Can't find that type...");
      };
      Controls.Add(btnDumpToFile);

      // Configure the TextBox.
      txtTypeName.Location = new System.Drawing.Point(13, 13);
      txtTypeName.Size = new System.Drawing.Size(341, 20);
      Controls.Add(txtTypeName);
    }
  }
}
```

To compile this Windows Forms application using a response file, create a file named `WinFormsClientApp.rsp`, and add the following commands:

```
/target:winexe
/out:WinFormsClientApp.exe
/r:CorLibDumper.dll
/r:System.Windows.Forms.dll
/r:System.Drawing.dll
WinFormsClientApp.cs
```

Now make sure this file is saved in the same folder as your example code, and supply this response file as an argument to `mcs` (using the `@` token):

```
msc @WinFormsClientApp.rsp
```

Finally, run your Windows Forms application using `Mono`:

```
mono WinFormsClientApp.exe
```

Figure B-8 shows a possible test run.

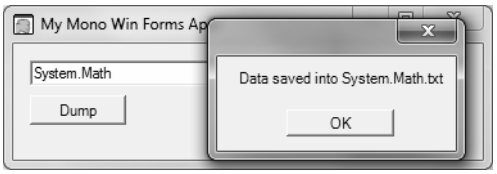

Figure B-8. *A Windows Forms Application Built Using Mono*

Executing Your Windows Forms Application Under Linux

So far, this appendix has shown you how to create a few assemblies that you could also have compiled using the Microsoft .NET Framework 4.0 SDK. However, the importance of Mono becomes clear when you view Figure B-9, which shows the same exact Windows Forms application running under SuSe Linux. Notice how the Windows Forms application has taken on the correct look and feel of my current desktop theme.

Figure B-9. *Executing the Windows Forms Application Under SuSe Linux*

■ **Source Code** You can find the `CorLibDumper` project under the Appendix B subdirectory.

The preceding examples shows how you can compile and execute the same exact C# code shown during this appendix on Linux (or any OS supported by Mono) using the same Mono development tools. In fact, you can deploy or recompile any of the assemblies created in this text that do not use .NET 4.0 programming constructs to a new Mono-aware OS and run them directly using the `Mono` runtime utility. Because all assemblies contain platform-agonistic CIL code, you do not need to recompile the applications whatsoever.

Who is Using Mono?

In this short appendix, I've tried to capture the promise of the Mono platform in a few simple examples. Granted, if you intend only to build .NET programs for the Windows family of operating systems, you might not have encountered companies or individuals who actively use Mono. Regardless, Mono is alive and well in the programming community for Mac OS X, Linux, and Windows.

For example, navigate to the `/Software` section of the Mono website:

```
http://mono-project.com/Software
```

At this location, you can find a long-running list of commercial products built using Mono, including development tools, server products, video games (including games for the Nintendo Wii and iPhone), and medical point-of-care systems.

If you take a few moments to click the provided links, you will quickly find that Mono is completely equipped to build enterprise-level, real-world .NET software that is truly cross-platform.

Suggestions for Further Study

If you have followed along with the materials presented in this book, you already know a great deal about Mono, given that it is an ECMA-compatible implementation of the CLI. If you want to learn more about Mono's particulars, the best place to begin is with the official Mono website (`www.mono-project.com`). Specifically, you should examine the page at `www.mono-project.com/Use`, which serves an entry point to a number of important topics, including database access using ADO.NET, web development using ASP.NET, and so on.

I have also authored some Mono-centric articles for the DevX website (`www.devx.com`) that you might find interesting:

- "Mono IDEs: Going Beyond the Command Line": This article examines many Mono-aware IDEs.
- "Building Robust UIs in Mono with Gtk#": This article examines building desktop applications using the GTK# toolkit as an alternative to Windows Forms.

Finally, you should familiarize yourself with Mono's documentation website (`www.go-mono.com/docs`). Here you will find documentation on the Mono base class libraries, development tools, and other topics (see Figure B-10).

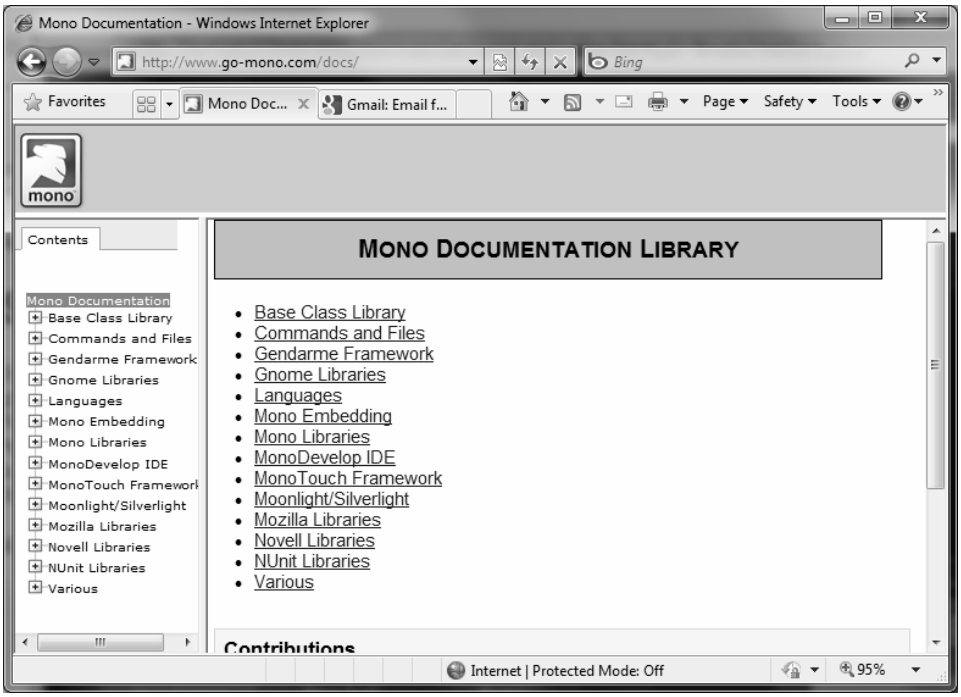

Figure B-10. *The Online Mono Documentation*

■ **Note** The website with Mono's online documentation is community supported; therefore, don't be too surprised if you find some incomplete documentation links! Given that Mono is an ECMA-compatible distribution of Microsoft .NET, you might prefer to use the feature-rich MSDN online documentation when exploring Mono.

Summary

The point of this appendix was to provide an introduction to the cross-platform nature of the C# programming language and the .NET platform when using the Mono framework. You have seen how Mono ships with a number of command-line tools that allow you to build a wide variety of .NET assemblies, including strongly named assemblies deployed to the GAC, Windows Forms applications, and .NET code libraries.

You've also learned that Mono is not fully compatible with the .NET 3.5 or .NET 4.0 programming APIs (WPF, WCF, WF, or LINQ) or the C# 2010 language features. Efforts are underway (through the Olive project) to bring these aspects of the Microsoft .NET platform to Mono. In any case, if you need to build .NET applications that can execute under a variety of operating systems, the Mono project is a wonderful choice for doing so.