CHAPTER 40
Using Mono

IN THIS CHAPTER

- Why Use Mono?
- MonoDevelop
- Building on Mono's Libraries
- References

Although Microsoft intended it for Windows, the Microsoft .NET platform has grown to encompass many other operating systems. No, this isn't a rare sign of Microsoft letting customers choose which OS is best for them. Instead, the spread of .NET is because of the Mono project, which is a free re-implementation of .NET available under the GPL license.

Because of the potential for patent complications, it took most distros a long time to incorporate Mono, but it's here now and works just fine. What's more, Mono supports both C# and Visual Basic .NET, as well as the complete .NET 1.0 and 1.1 Frameworks (and much of the 2.0 Framework, too), making it easy to learn and productive to use.

There were some fears that Mono may cease to exist. In early May 2011, Attachmate bought Novell along with its SUSE Linux distribution. Novell headed up the Mono project. One of the first things that Attachmate did after the acquisition was cancel the Mono project. Then they laid off all the Mono developers, about 30 people.

On May 16, 2011, the lead developer of Mono, Miguel de Icaza, announced on his blog the creation of a new company called Xamarin (http://xamarin.com). The new company was created to continue Mono development and to build Mono applications. Many, if not most, of the original developers working on Mono at Novell are now employed by Xamarin, and Mono 2.10 is installed by default in Ubuntu 11.10, along with several default applications that are written in Mono. On July 18, 2011, it was announced by SUSE that an intellectual property agreement and perpetual license had been extended to Xamarin to continue to use and develop Mono as a platform and

products that use Mono. No one knows what the future holds for Mono, but at least we know it has not yet been abandoned. In August 2011, Xamarin made its first official Mono release.

Why Use Mono?

Linux already has numerous programming languages available to it, so why bother with Mono and .NET? Here are my top five reasons:

- .NET is "compile once, run anywhere"; this means you can compile your code on Linux and run it on Windows, or the reverse.
- Mono supports C#, which is a C-like language with many improvements to help make it object-oriented and easier to use.
- .NET includes automatic garbage collection to remove the possibility of memory leaks.
- .NET uses comes with built-in security checks to ensure that buffer overflows and many types of exploits are a thing of the past. Mono uses a high-performance just-in-time compiler to optimize your code for the platform on which it's running. This lets you compile it on a 32-bit machine, and then run it on a 64-bit machine and have the code dynamically recompiled for maximum 64-bit performance.

At this point, Mono is probably starting to sound like Java, and indeed it shares several properties with it. However, Mono has the following improvements:

- The C# language corrects many of the irritations in Java, while keeping its garbage collection.
- .NET is designed to let you compile multiple languages down to the same bytecode, including C#, Visual Basic .NET, and many others.
- Mono even has a special project (known as IKVM) that compiles Java source code down to .NET code that can be run on Mono.
- Mono is completely open source!

Whether you're looking to create command-line programs, graphical user interface apps, or even web pages, Mono has all the power and functionality you need.

Mono is made up of several components, including a C# compiler, the Mono Runtime, the Base Class Library, and the Mono Class Library.

The C# compiler includes all of the features to compile C# 1.0, 2.0, and 3.0 (ECMA) code. The compiler is able to compile itself, is fast, and includes a test suite. The C# compiler gives you your pick of several services that applications may consume, as follows:

- **`mcs`**—References the 1.0-profile libraries and supports C# 1.0 and C# 3.0, minus generics and any features that depend on generics, and is called `mono-cms` in the Ubuntu software repositories.

- **gmcs**—References the 2.0-profile libraries and supports the full C# 3.0 language, is called `mono-gcms` in the Ubuntu software repositories, and is installed by default in Ubuntu 10.10.
- **smcs**—References the 2.1-profile libraries, supports the full C# 3.0 language, and is used for creating Moonlight (the Mono version of Silverlight) applications. It is not available in the Ubuntu software repositories.
- **dmcs**—Targets the C# 4.0 language and is not available in the Ubuntu software repositories.

The first three in that list are all the same compiler, but with different default settings. The last one listed is new and will become the default in Mono 2.8.

The Mono Runtime uses the ECMA *Common Language Infrastructure (CIL)*. This is a runtime similar to the Java Virtual Machine. It provides a *just-in-time (JIT)* compiler, an *ahead-of-time (AOT)* compiler, a library loader, a garbage collector, a threading system, and interoperability functionality. The execution engine, `mono`, can be used as a standalone process. By itself, it runs as a JIT or AOT code generator. With `--full-aot`, it can run in fully static mode with no JIT involvement. You can reuse your existing C and C++ code and extend it with Mono either with scripting or by embedding Mono into applications.

The Mono Base Class Library provides a set of foundational classes that are compatible with Microsoft's .NET Framework. It is extended by the Mono Class Library that provides additional functions that are especially useful when building Linux applications, such as classes for Gtk+, LDAP, and POSIX.

Mono runs on many platforms, not just Linux. It is designed as a free-software implementation of Microsoft's C# and can also run on Windows, Mac OS X, BSD, Solaris, Nintendo Wii, Sony PlayStation 3, and the Apple iPhone. It works on many different types of processors, including the common x86 and x86-64 that Ubuntu targets, and also others such as PowerPC, SPARC, ARM, Alpha, and more. The *Common Language Runtime (CLR)* that comes with Mono allows you to choose other programming languages, write your code using them, and then run the programs in Mono (see http://mono-project.com/Languages).

MonoDevelop

- Mono should already be installed on your system, but it is installed only for end users rather than for developers. You need to install a couple more packages to make it usable for programming. Install `mono-complete` and `monodevelop` from the Ubuntu repositories.
- These give you the basics to do Mono development.

If you want to do other exciting things with Mono, lots of Mono-enabled libraries are available. Use Synaptic at System, Administration, Synaptic Package Manager to search the Ubuntu software repositories for "sharp" to bring up the list of .NET-enabled libraries that

you can use with Mono; the suffix is used because C# is the most popular .NET language, indeed the flagship language in the .NET fleet, written especially for .NET. In this list, you'll see things such as gnome-sharp2 and gtk-sharp2—we recommend you at least install the gtk-sharp2 libraries because these are used to create graphical user interfaces for Mono.

Open MonoDevelop from the menu at Applications, Programming, MonoDevelop.

TIP

You don't have to use MonoDevelop to write your code, but it helps—syntax highlighting, code completion, and drag-and-drop GUI designers are just a few of its features.

When MonoDevelop has loaded, open File, New, Solution. From the left of the window that appears, choose C#, then Console Project. Give it a name and choose a location to save it, as in Figure 40.1. When you're ready, click Forward.

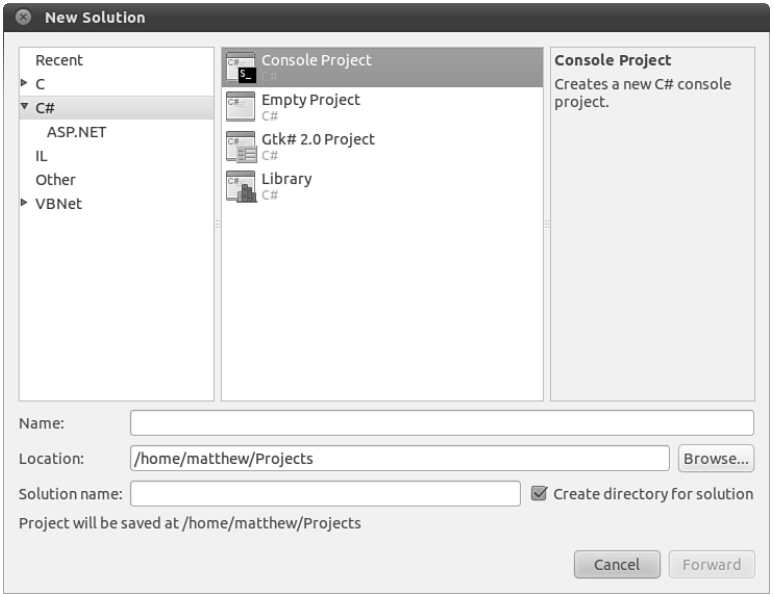

FIGURE 40.1 MonoDevelop ships with a number of templates to get you started, including one for a quick Console Project.

This is a simple command-line project. For our example, select UNIX Integration as the only feature to include in Figure 40.2 and click OK. Notice that this generates a launch script by default and allows you to choose to create a .desktop file if desired. You can also create packaging as an archive of binaries or a tarball and enable support for GTK#. We don't need either of these for our example.

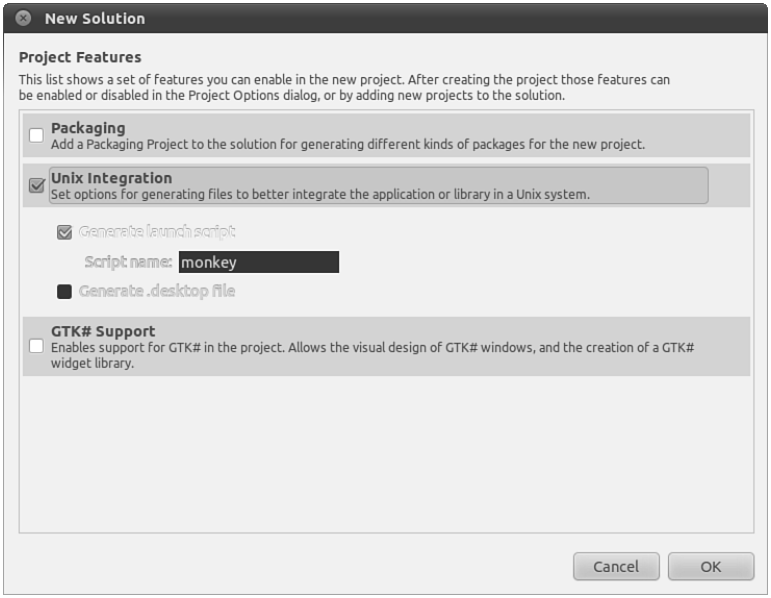

FIGURE 40.2 Choose the project features you want to enable in your project.

The default Console Project template creates a program that prints a simple message to the command line: the oh-so-traditional "Hello World!" Click Build, Build HelloWorld (or F7) to build the project (see Figure 40.3). Click Run, Run (or Ctrl+F5) to run the program. A terminal window will appear to show you the output of your program, as shown in Figure 40.4.

You can also run and debug at the same time using Run, Debug (or F5). If there are errors in the program code, you are notified, as shown in Figure 40.5. I created a simple error by removing a letter from the end of a word. If there are no errors, the program runs as it did when we used Run, Run previously.

The Structure of a C# Program

As you can guess from its name, C# draws heavily on C and C++ for its syntax, but it borrows several ideas from other languages that its creator, Hejlsberg, had experience with, such as Java, Turbo Pascal, Delphi, Python, Icon, Smalltalk, and Modula-2 and 3. C# is object-oriented. In C#, you define a class, which is then instantiated by calling its constructor.

`Main()` is the default entry point for an executable assembly in MonoDevelop and Visual Studio .NET. To be able to draw on some of the .NET Framework's many libraries, you need to add `using` statements at the top of your files—by default, only `using System;` is included, enabling you to access the console to write your Hello World! message.

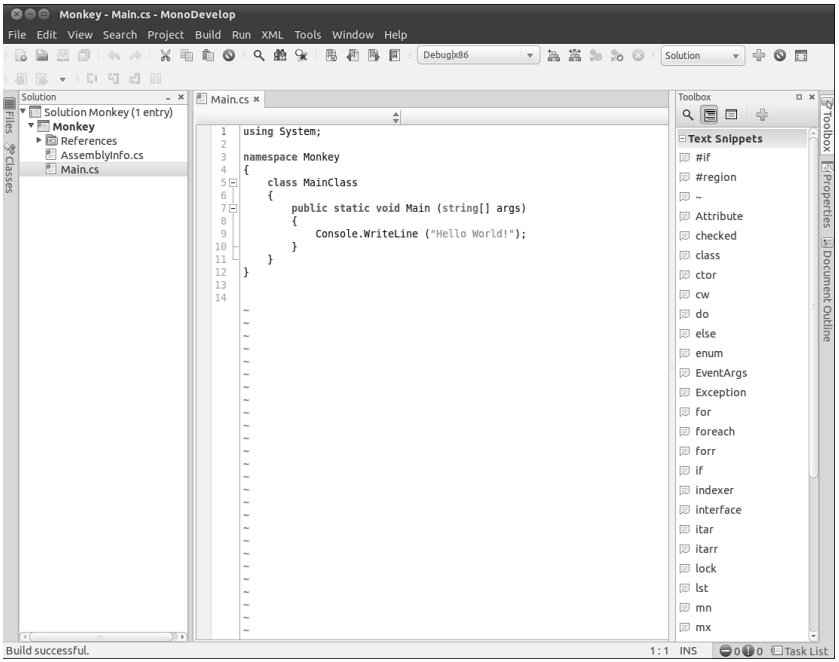

FIGURE 40.3 MonoDevelop with HelloWorld loaded, showing a successful build notice at the bottom.

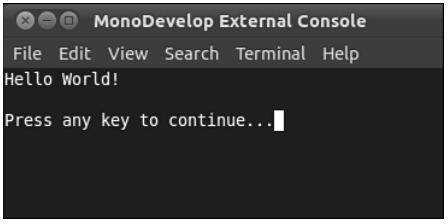

FIGURE 40.4 A successful run of HelloWorld in a terminal.

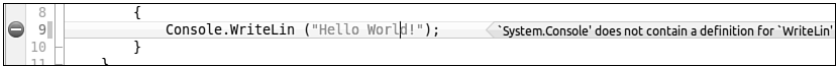

FIGURE 40.5 An error message points out a problem when running in debug mode.

If you have experience with C or C++, you will notice that there are no header files in C#; your class definition and its implementation are all in the same file. You might also have noticed that the `Main()` method accepts the parameter `"string[] args"`, which means we will be passing arguments from the command line and is C#-speak for "an array of

strings." C never had a native "string" data type, whereas C++ acquired it rather late in the game, and so both languages tend to use the antiquated `char*` data type to point to a string of characters. In C#, `"string"` is a data type all its own, and comes with built-in functionality such as the capability to replace substrings, the capability to trim off white-space, and the capability to convert itself to uppercase or lowercase if you want it to. Strings are also Unicode friendly out of the box in .NET, so that's one fewer thing for you to worry about.

The final thing you might have noticed—at least, if you had looked in the directory where MonoDevelop placed your compiled program (usually `/path/to/your/project/bin/Debug`)—is that Mono uses the Windows-like `.exe` file extension for its programs, because Mono aims to be 100-percent compatible with Microsoft .NET, which means you can take your Hello World program and run it unmodified on a Windows machine and have the same message printed out.

Printing Out the Parameters

We're going to expand this little program by having it print out all the parameters passed to it, one per line. In C#, this is—amazingly—just one line of code. Add this just after the existing `Console.WriteLine()` line in your program:

```
foreach (string arg in args) Console.WriteLine(arg);
```

The `foreach` keyword is a special kind of loop designed to iterate over any collection, in this case an array. The for loop exists in C#, and lets you loop a certain number of times; the `while` loop exists, too, and lets you loop continuously until a conditional is no longer true or a break statement is encountered—the conditional controls the loop and a break statement will allow an exit if desired or included. But the `foreach` loop is designed to loop over a collection from the start to the end. You get each value as a variable of any type you want—the preceding code says `string arg in args`, which means "for each array element in args, give it to me as the variable arg of type string." Because `args` is already an array consisting of many `strings`, no data type conversion takes place here—but it could if you wanted to convert classes or do anything of the like. One reason you might use a for loop over a `foreach` loop is if you need to determine an index into a collection, as in finding a minimum or maximum value. There is no index available for the current iteration in a `foreach` loop, much like it is not available in a `while` loop.

After you have each individual argument, call `WriteLine()` to print it out. This works whether there's one argument or one hundred arguments—or even if there are no arguments at all (in which case, the loop doesn't execute at all).

Creating Your Own Variables

You can have anonymous variable types and `var` variable declarations in C#. You can also specify variable types with great precision. Most of the time, you can use `int` and leave C# to work out the details. Let's modify our program to accept two parameters, add them

together as numbers, and then print the result. This gives you a chance to see variable definitions, conversion, and mathematics, all in one. Edit the `Main()` method to look like this:

```
using System;

namespace AddNumbers
{
        class MainClass
       {
                public static void Main (string[] args)
                 {
                        int num1 = int.Parse(args[0]);
                        int num2 = int.Parse(args[1]);
                        Console.WriteLine("Sum of two parameters is: " + (num1 +
num2)");
                 }
        }
}
```

Run this new code using Run, Run With, Custom Parameter. As you can see, each variable is declared with a type (`int`) and a name (`num1` and `num2`) so that C# knows how to handle them. The `args` array contains strings, so we convert the strings to integers with the `int.Parse()` method. Finally, the actual addition of the two strings is done at the end of the method, while they are being printed out. Note how C# is clever enough to have integer + integer be added together (in the case of `num` + `num2`), whereas string + integer attaches the integer to the end of the string (in the case of "Sum of two parameters is:" + the result of `num1` + `num2`). This isn't by accident: C# tries to convert data types cleverly, and warns you only if it can't convert a data type without losing some data. For example, if you try to treat a 64-bit integer as a 32-bit integer, it warns you because you might be throwing a lot of data away. You can use `int.TryParse()` to attempt parsing conditionally, and it would also return a Boolean response indicating success or failure. In addition, you can also use checked conversions to perform operations and have the compiler verify that they do not cause overflow (essentially meaning the variable could not represent the value).

Adding Some Error Checking

Right now, your program crashes in a nasty way if users don't provide at least two parameters. The reason for this is that we use `arg[0]` and `arg[1]` (the first and second parameters passed to your program) without even checking whether *any* parameters were passed in. This is easily solved: `args` is an array, and arrays can reveal their size. If the size doesn't match what you expect, you can bail out.

Add this code at the start of the `Main()` method:

```
if (args.Length != 2) {
   Console.WriteLine("You must provide exactly two parameters!");
   return;
}
```

The new piece of code in there is return, which is a C# keyword that forces it to exit the current method. Because `Main()` is the only method being called, this has the effect of terminating the program because the user didn't supply two parameters. You could also specify that the `Main()` method have a return type of int so that an exit code would be returned instead of returning without any feedback. This would be especially helpful if the executable was to be used in a command-line tool chain by another tool that needs feedback after your tool runs.

Using the `Length` property of `args`, it is now possible for you to write your own `Main()` method that does different things, depending on how many parameters are provided. To do this properly, you need to use the `else` statement and nest multiple `if` statements, like this:

```
if (args.Length == 2) {
   /// whatever...
} else if (args.Length == 3) {
   /// something else
} else if (args.Length == 4) {
   /// even more
} else {
   /// only executed if none of the others are
}
```

Building on Mono's Libraries

Ubuntu ships with several Mono-built programs, such as Tomboy and gBrainy. It also comes with a fair collection of .NET-enabled libraries, some of which you probably already installed earlier. The nice thing about Mono, like many other object-oriented languages, is that it lets you build on these libraries really easily: You just import them with a `using` statement and then get started.

To demonstrate how easy it is to build more complicated Mono applications, we're going to produce one using Gtk#, the GUI toolkit that is the standard for GNOME development.

Creating a GUI with Gtk#

Gtk# was included with GNOME by default for the first time in GNOME 2.16, but it had been used for a couple of years before that and so was already mature. MonoDevelop comes with its own GUI designer called GTK# Designer, which lets you drag and drop GUI elements onto your windows to design them.

To get started, in MonoDevelop, go to File, New, Solution, choose C#, and then choose Gtk# 2.0 project. Call it **GtkTest**, and deselect the box asking MonoDevelop to make a separate directory for the solution. Click Forward. In the New Solution Project Features window, all you need is already selected, so click OK.

You'll find that `Main.cs` contains a little more code this time because it needs to create and run the Gtk# application. However, the actual code to create your GUI is in the User

Interface section in the left pane. If you open that group, you'll see Stock Icons and MainWindow. Double-click MainWindow to bring up MonoDevelop's GUI designer.

Click Toolbox on the left of the screen to drag and drop the different window widgets onto your form to see what properties they have.

For now, drag a button widget onto your form. It automatically takes up all the space on your window. If you don't want this to happen, try placing one of the containers down first, and then putting your button in there. For example, if you want a menu bar at the top, then a calendar, and then a status bar, you ought to drop the VBox pane onto the window first, and then drop each of those widgets into the separate parts of the VPane, the main GUI creation workspace.

Your button displays the text GtkButton and uses the variable name button1 by default. Click to select it, and then click Properties on the right and look for Label in the Button Properties group. Change the label to **Hello**. Just at the top of the Properties pane is a tab saying Properties (where you are right now), and another saying Signals. Signals are the events that happen to your widgets, such as the mouse moving over them, someone typing, or, of interest to us, when your button has been clicked. Look inside the Button Signals group for Clicked and double-click it. MonoDevelop automatically switches you to the code view, with a pre-created method to handle button clicks.

Type this code into the method:

```
button1.Label = "World!";
```

If you want to change the button's variable name, you do so from the same Properties tab using the first entry, Name. Change the Name value, and your variable's name is changed. Press F5 to compile and run, and try clicking the button.

References

- **www.mono-project.com/**—The home page of the Mono project is packed with information to help you get started. You can also download new Mono versions from here, if there's something you desperately need.
- **www.monodevelop.com/**—The MonoDevelop project has its own site, which is the best place to look for updates.
- **www.icsharpcode.net/OpenSource/SD/**—MonoDevelop started life as a port of SharpDevelop. If you happen to dual-boot on Windows, this might prove very useful to you.
- **http://msdn.microsoft.com/vcsharp/**—We don't print many Microsoft URLs in this book, but this one is important: It's the home page of their C# project, which can be considered the spiritual home of C# itself.

- Jesse Liberty's *Programming C#* (O'Reilly, ISBN 0-596-00699-3) is compact, it's comprehensive, and it's competitively priced.
- If you're very short on time and want the maximum detail (admittedly, with rather limited readability), you should try *The C# Programming Language*, which was co-authored by the creator of C#, Anders Hejlsberg (Addison-Wesley, ISBN: 0-321-33443-4).
- For a more general book on the .NET framework and all the features it provides, you might find *.NET Framework Essentials* (O'Reilly, ISBN: 0-596-00505-9) to be useful.