

Mono

A Developer's Notebook™

Edd Dumbill
Niel M. Bornstein

- Gtk#
- MonoDevelop
- Web Services
- IKVM

O'REILLY®

Core .NET

By now you've become familiar with the C# language, and you've seen some of its differences and similarities to Java and C/C++. We've alluded to some of the powerful *Framework Class Library* (FCL) classes, but we haven't really gone into depth on them.

In this chapter, you'll be introduced to the FCL and some of its most commonly used classes, including those that let you work with files, strings, regular expressions, collections, assemblies, process control, and threading. And, as a bonus, you'll get your first look at NUnit, the .NET unit testing framework.

Work with Files

File I/O is an important part of any class library, and the FCL contains a complete and powerful set of file access classes. In this lab, you'll see how to manipulate files and directories with Mono.

How do I do that?

You need to learn three basic classes in the FCL to manage files. They are `File`, `TextReader`, and `TextWriter`. Example 3-1 shows the source for a class that uses these three classes to create a file, write to it, read from it, and set some of the file's attributes.

Example 3-1. 01-files/FileCreator.cs

```
// 03-keyfunc/01-files
using System;
using System.IO;

public class FileCreator {
```

In this chapter:

- "Work with Files"
- "Manage String Data"
- "Search Text with Regular Expressions"
- "Manage Collections of Data"
- "Work with Assemblies"
- "Start and Examine Processes"
- "Multitask with Threads"
- "Test Your C# Code"

Reading, writing, and arithmetic may be elementary skills, but Mono makes them even easier.

Example 3-1. 01-files/FileCreator.cs (continued)

There are attributes, attributes, and attributes. In "Add Metadata to Your Types" in Chapter 2, we told you about attributes in C#. In Chapter 6 we'll talk about XML attributes. This lab features file attributes, indicating metadata such as whether the file is to be archived, and whether it's compressed, encrypted, hidden, and others.

```
public static void Main(string [] args) {
    string file = args[0];

    if (File.Exists(file)) {
        Console.WriteLine("File {0} exists with attributes {1}," +
            "created at {2}", file, File.GetAttributes(file),
            File.GetCreationTime(file));
    } else {
        using (TextWriter writer = File.CreateText(file)) {
            writer.WriteLine("Greetings from Mono!");
        }
        File.SetAttributes(file,
            File.GetAttributes(file) | FileAttributes.ReadOnly |
            FileAttributes.Temporary);
    }

    using (TextReader reader = File.OpenText(file)) {
        Console.WriteLine(reader.ReadToEnd());
    }
}
```

You should always remember to close any files you open. Luckily, TextReader and TextWriter both implement IDisposable, so you can wrap them in the using statement to make sure they get closed. This using is different than the one used to import namespaces. Here's how using works in this context: classes that implement IDisposable have a Dispose method. In Example 3-1, this code:

```
using (TextReader reader = File.OpenText(file)) {
    Console.WriteLine(reader.ReadToEnd());
}
```

translates into:

```
{
    TextReader reader = File.OpenText(file);
    try {
        Console.WriteLine(reader.ReadToEnd());
    } finally {
        if (reader != null) {
            ((IDisposable)reader).Dispose();
        }
    }
}
```

Most classes that use system resources implement IDisposable, and care should be taken to always use the using statement to guarantee a timely release of those resources.

How it works

The `File` class, which you previously saw in Example 2-7 of Chapter 2, contains methods that you can use to access files and information about them. Another class, `Stream`, lets you access any resource (including files, memory, and network sockets) as a stream of bytes. Finally, `TextReader` and `TextWriter` let you read and write files as text.

Some of the more useful methods of `File` are `Exists()`, `Create()`, and `Delete()`, and several methods to open a file. Table 3-1 describes the file-creating and file-opening methods, their parameters and return values, and what they do.

Table 3-1. Methods to create and open files

Method	Parameters	Returns	Description
<code>Create()</code>	string, int	<code>FileStream</code>	(Two overloads) Creates the file with the given buffer size for reads and writes.
<code>CreateText()</code>	string, int	<code>StreamWriter</code>	Creates the file.
<code>Open()</code>	string, <code>FileMode</code> , <code>FileAccess</code> , <code>FileShare</code>	<code>FileStream</code>	(Three overloads) Opens the file for reading and/or writing, depending on the <code>FileMode</code> and <code>FileAccess</code> parameters. Other processes' access to the file is limited by the <code>FileShare</code> parameter.
<code>OpenRead()</code>	string	<code>FileStream</code>	Opens the file for reading.
<code>OpenText()</code>	string	<code>StreamReader</code>	Opens the file for reading.
<code>OpenWrite()</code>	string	<code>FileStream</code>	Opens the file for writing.

In addition to the methods listed in Table 3-1, the `GetAttributes()` method returns a `FileAttributes` enum, which contains information about the file. And the `SetAttributes()` method sets the file's attributes. There are also `GetLastAccessTime()` and `SetLastAccessTime()`, `GetLastWriteTime()`, `SetLastWriteTime()`, and others. See monodoc for more information.

The `FileAttributes` enum was designed to map to the Windows filesystem, so some of its values don't make sense in a Linux or Mac OS X Mono context. You can use the `Syscall` class from the `Mono.Posix` namespace to manipulate file permissions through the `chmod()` and `chown()` methods, as shown in Example 3-2.

*Mono.Posix.
FileMode is a
completely
different class
from System.IO.
FileMode,
described in
Table 3-1. Don't
set them
confused!*

Example 3-2. 01-files/Syscall.cs

```
// 03-keyfunc/01-files
using Mono.Posix;

public class ChmodTest {
    public static void Main(string [] args) {
        string file = args[0];
        Syscall.chmod(file, (FileMode)0x777);

        int uid = Syscall.getuid();
        int gid = Syscall.getegid();
        Syscall.chown(file, uid, gid);
    }
}
```

To compile Example 3-2, you must reference the Mono.Posix DLL with the following command line:

```
$ mcs -r:Mono.Posix Syscall.cs
```

Syscall's methods map very closely to the POSIX functions with the same name. `chmod()` and `chown()` do exactly what their POSIX (and command line) namesakes do. And there are other POSIX functions available, like `open()`, `close()`, and `creat()`, in addition to `getuid()` and `getegid()`.

The code in Example 3-2 takes a filename on the command line and changes its permissions to `0x777` (`rwXrwXrwX`), full read, write, and execute access, and changes its owner to the current process's user ID and effective group ID.

What about ...

...Reading binary data from a file? `TextReader` and its subclasses can only be used to read text from files, so you must use a different class. Recall that the `File` class' `Open()` and `Create()` methods return a `FileStream`. The `Stream` class has several subclasses, including `FileStream`, `MemoryStream`, and `NetworkStream`, designed to provide byte-level access to resources.

`Stream`'s methods include `Read()` and `Write()`, as well as methods and properties like `Seek()`, `Flush()`, `Position`, and `Length`. These methods use byte arrays as buffers to store data as it is read and written rather than the strings that `TextReader` and `TextWriter` use.

There are also two classes analogous to the `TextReader` and `TextWriter`. Called `BinaryReader` and `BinaryWriter`, respectively, they let you read and write data of a particular type from a binary file. `BinaryReader`, for

example, has methods called `ReadByte()` and `ReadDouble()` that read the appropriate amount of data from the underlying `Stream` and return it in an instance of the appropriate type.

`Stream` also allows asynchronous access to resources through the `BeginRead()` and `EndRead()`, and `BeginWrite()` and `EndWrite()` methods.

Where to learn more

Asynchronous I/O is covered in the *.NET Framework Developer's Guide* at <http://msdn.microsoft.com/library/en-us/cpguide/html/cpconASynchronousFileIO.asp>.

Almost every program has to deal with strings in one way or another.

Manage String Data

Strings are first-class objects in C#. They can be created just like any other type, but you will do well to use them wisely.

How do I do that?

Example 3-3 shows how strings are created and manipulated using the `string` class.

Example 3-3. 02-strings/Strings.cs

```
// 03-keyfunc/02-strings
using System;

public class Strings {
    public static void Main(string [] args) {
        string s1 = "a string!";
        string s2 = @"a string with \escaped characters,
including a carriage return";
        string s3 = new String('a', 4);
        string s4 = s3 + s1;
        string s5 = s4.Replace('a', 'b');

        Console.WriteLine(s1);
        Console.WriteLine(s2);
        Console.WriteLine(s3);
        Console.WriteLine(s4);
        Console.WriteLine(s5);
    }
}
```

`System.String`, like `Int32`, `Byte`, `Char`, and other primitive types in the `System` namespace, can be referred to by an alias. For `String`, that's `string`.

Compiling and running this program, here's what you'll see:

```
$ mcs Strings.cs
$ mono Strings.exe
a string!
a string with \escaped characters,
including a carriage return
aaaa
aaaaa string!
bbbbbb string!
```

Going over these one at a time, here's how they work. The first string, `s1`, is assigned to the literal value "a string!". Simple enough.

The second string, `s2`, is prefixed with the `@` character. This `@`-quoted (literal) string contains an embedded backslash and a carriage return. Special characters such as these typically need to be escaped using the `\\` and `\n` character sequences. By `@`-quoting the string, any characters that normally need to be escaped can be put in the string literally.

`s3` demonstrates one of the `string` class' overloaded constructors. This one takes a `char` and an `int`, and creates a string containing the `char` value repeated by the `int` number of times.

`s4` demonstrates the concatenation of two strings. In this case, a new string is created containing the concatenated values of `s2` and `s3`.

Finally, `s5` shows how you can use the `string` class' `Replace()` method to return a new string instance with every instance of the character `a` replaced with the character `b`. Notice that the original string is not changed, because the `string` class is immutable.

Of course, concatenating strings with the `+` operator is not the most efficient use of memory. Each original string has to be allocated, and then a new string has to be allocated with the concatenated contents of the two strings. And because the `string` class is immutable, you can't just set some character within the string directly, which means you have to allocate another string to return every time. Wouldn't it be nice to have a way to do these things less expensively?

Sure it would. Enter `StringBuilder`. A member of the `System.Text` namespace, `StringBuilder` provides a way to build strings that you can modify and format on the fly.

We can replace the string `s4` with a `StringBuilder` thus:

```
StringBuilder s4 = new StringBuilder();
s4.Append(s3);
s4.Append(s1);

Console.WriteLine(s4.ToString());
```

Unicode? Of course! Strings in Mono are stored in Unicode internally. To embed a Unicode character within a string, use the format `\unnnn`, where `nnnn` is a four-digit number, or `\unnnn\unnnn` for 8-digit Unicode characters.

Because the `StringBuilder` is not a string, you must call its `ToString()` method to get a string representation of it.

Or, you could use one of the `StringBuilder`'s overloaded constructors to condense the code, like this:

```
StringBuilder s4 = new StringBuilder(s3);
s4.Append(s1);
```

And, because the `Append()` method returns its `StringBuilder` instance, you can chain the various calls like this:

```
StringBuilder s4 = new StringBuilder(s3).Append(s1);
```

What about ...

...Formatting strings? You may be accustomed to `printf()` and its `%s` and `%d` format strings. `Mono` handles composite formatting with the following syntax:

```
string s = string.Format("{0} {1} {2:C} {3:N4}",
    DateTime.Now, 'a', 5.95, 1);
```

After this assignment, the string `s` would have the value `Saturday, 24 April 2004 12:18:35 a $5.95 1.0000`.

The `Format()` method has the following signature:

```
public static string Format(string format, params object [] args);
```

The `params` keyword is the C# way of indicating a variable argument list. Although the parameter type is `object []`, you don't have to literally instantiate a new array of objects to hold the arguments. Instead, you can just specify them one by one, and the compiler will create an array on the fly to be passed to the method. Each parameter's `ToString()` method will be called to produce the output.

The string formatting rules are simple. Each element in the `params` array can be referenced by its ordinal, so the format `{0}` refers to array element 0, `{1}` refers to array element 1, and so on. In addition to the ordinal, you can specify more detailed formatting rules for individual objects, such as the number of digits.

`StringBuilder` also lets you format strings. The `AppendFormat()` method has a similar signature:

```
public StringBuilder AppendFormat(string format, params object [] args);
```

Given a `StringBuilder`, you can append a formatted string to it like so:

```
StringBuilder s = new StringBuilder();
s.AppendFormat("{0} {1} {2:C} {3:N4}", DateTime.Now, 'a', 5.95, 1);
```

The exact string representation of a `DateTime` value and of numbers formatted with the `C` currency format specifier, should depend on your locale—and, of course, the date and time that you run the program.

Parse text with obscure, magical formulae that look like a baby sat down at the keyboard.

The hosts file will be located in different places in different operating systems. In Linux, Unix, and Mac OS X, it will be in /etc. On Windows, it will be in %WINDIR%\System32\drivers\etc. When constructing a path, use the Path.DirectorySeparatorChar property, which gives you the character that separates directories in a path, / for Unix, \ for Windows.

Search Text with Regular Expressions

Perl and shell programmers have long known the power of *regular expressions*, a language for describing patterns in sometimes excruciating detail. But there's no substitute for a regular expression when you need to separate a fixed- or variable-format chunk of text into its components.

This lab will demonstrate Mono's regular expression support.

How do I do that?

Example 3-4 shows a program that reads a *hosts* file and reports on its contents using regular expressions.

Example 3-4. 03-regex/ParseHosts.cs

```
// 03-keyfunc/03-regex
using System;
using System.Collections;
using System.IO;
using System.Text.RegularExpressions;

public class ParseHosts {
    public static void Main(string [] args) {
        string filename;
        if (Environment.OSVersion.ToString().StartsWith("Unix")) {
            filename = string.Format("{0}{1}etc{1}hosts",
                Path.DirectorySeparatorChar);
        } else {
            filename = string.Format("{0}{1}drivers{1}etc{1}hosts",
                Environment.GetFolderPath(Environment.SpecialFolder.System),
                Path.DirectorySeparatorChar);
        }

        if (!File.Exists(filename)) {
            Console.Error.WriteLine("{0} does not exist.", filename);
            Environment.Exit(1);
        }

        string text;
        using (TextReader reader = File.OpenText(filename)) {
            text = reader.ReadToEnd();
        }

        Regex regex =
            new Regex(@"(?<ip>(\d{1,3}\.){3}\d{1,3})\s+(?<name>(\S+))");

        MatchCollection matches = regex.Matches(text);
        foreach (Match match in matches) {
```

Example 3-4. 03-regex/ParseHosts.cs (continued)

```
        if (match.Length != 0) {
            Console.WriteLine("hostname {0} is mapped to ip address {1}",
                match.Groups["name"], match.Groups["ip"]);
        }
    }
}
```

My *hosts* looks like this:

```
##
# Host Database
#
# localhost is used to configure the loopback interface
# when the system is booting. Do not change this entry.
##
127.0.0.1        localhost
255.255.255.255 broadcasthost
::1             localhost

# reroute gracenote
64.71.163.204   cddb.cddb.org

# adware blockers
127.0.0.1       207-87-18-203.wsmg.digex.net
127.0.0.1       Garden.ngadcenter.net
127.0.0.1       Ogilvy.ngadcenter.net
...
```

Your hosts file may be very short or completely empty. Ours is full of tricks to keep ads from annoying us when we surf the web.

You can compile and run *ParseHosts.exe* as usual, and see the following output:

```
$ mcs ParseHosts.cs
$ mono ParseHosts.exe
hostname localhost is mapped to ip address 127.0.0.1
hostname broadcasthost is mapped to ip address 255.255.255.255
hostname cddb.cddb.org is mapped to ip address 64.71.163.204
hostname 207-87-18-203.wsmg.digex.net is mapped to ip address 127.0.0.1
hostname Garden.ngadcenter.net is mapped to ip address 127.0.0.1
hostname Ogilvy.ngadcenter.net is mapped to ip address 127.0.0.1
...
```

How it works

The `System.Text.RegularExpressions` namespace contains a suite of classes that can be used to search text using regular expressions. The main class, `Regex`, represents a single regular expression and can be used to determine whether a given string matches the expression, and if it does, returns a collection of `Match` instances.

Example 3-4 demonstrates the use of the `Regex`, `Match`, and `Group` classes. First, some housekeeping must be done to find the `hosts` file based on the operating system and to read the file into a string called `text`. Then, the regular expression work begins. We create an instance of the `Regex` class for a particular regular expression.

The C# regular expression language is similar to others that you may be familiar with, although there are a few details unique to C#. This pattern, `(?<ip>(\d{1,3}\.){3}\d{1,3})\s+(?<name>(\S*))`, will match a host entry from the `hosts` file. The pattern breaks down like this:

```
(?<ip>(\d{1,3}\.){3}\d{1,3})
```

Match a group of one to three decimal digits `(\d{1,3})` followed by a `.` `(\.)`, repeated three times `{3}`, and then one to three additional decimal digits `(\d{1,3})`. The `?<ip>` notation and the outer parentheses indicate that the whole thing should be grouped and called `ip`.

```
\s+
```

Match one or more whitespace characters, and throw them away.

```
(?<name>(\S+))
```

Match any number of any nonwhitespace character. Group the whole thing and call it `name`.

The `Matches()` method returns a `MatchCollection`. This collection contains a `Match` instance for each line in the text variable that matches the regular expression.

Ok, this is not a perfect regular expression for a host entry. We haven't considered that each piece of the dotted-decimal IP address can be no greater than 255, and that there are further limitations based on whether it's a class A, B, or C address, but we're just trying to get the point across.

The `Match` class has a `Groups` property that returns a `GroupCollection`. The `GroupCollection` contains an instance of `Group` for each group in the regular expression. To obtain each group, the program iterates over all the `Match` instances, retrieves the `Groups` by name ("name" and "ip"), and then prints a message about the host entry to standard output.

Where to learn more

A good reference for regular expressions in general is *Mastering Regular Expressions* by Jeffrey E. F. Friedl (O'Reilly). For .NET regular expressions, which are slightly different, the MSDN documentation is available at <http://msdn.microsoft.com/library/en-us/cpguide/html/cpconCOMRegularExpressions.asp>.

There's a static version of `Matches()` that you can use if you're only going to call it once. The version we used in the example may be more efficient if you're going to call it multiple times.

A group is anything that has parentheses around it, not just if it's got a name. We're only interested in the named groups here, though.

Manage Collections of Data

Because you can never have too much of a good thing.

Collection types in the .NET Framework are very powerful. Besides the array type we introduced in “Model the Behavior of Real-World Things” in Chapter 2, there is a whole slew of more complex and specialized types to handle almost every case where you need a collection of objects. And you can extend a base class to make your own custom collection.

In this lab, we’ll introduce the many ways of working with collections in Mono.

How do I do that?

Example 3-5 shows a class that uses some of the many .NET collection types.

Example 3-5. 04-collections/PrintEnvironment.cs

```
// 03-keyfunc/04-collections
using System;
using System.Collections;

public class PrintEnvironment {
    public static void Main(string [] args) {
        IDictionary variables = Environment.GetEnvironmentVariables();
        Console.WriteLine(variables.GetType());

        ICollection variableNames = variables.Keys;
        Console.WriteLine(variableNames.GetType());

        foreach (string variableName in variableNames) {
            Console.WriteLine("{0}={1}", variableName, variables[variableName]);
        }
    }
}
```

If you compile and run *PrintEnvironment.exe*, you should see something like this:

```
$ mcs PrintEnvironment.cs
$ mono PrintEnvironment.exe
MANPATH=/sw/share/man:/usr/share/man:/usr/local/man:/usr/X11R6/man
VISUAL=/usr/bin/vi
TERM_PROGRAM=Apple_Terminal
TERM_PROGRAM_VERSION=100
OLDPWD=/Users/niel
__CF_USER_TEXT_ENCODING=0x1F5:0:0
XML_CATALOG_FILES=/sw/etc/xml/catalog
INFOPATH=/sw/share/info:/sw/info:/usr/share/info
LOGNAME=niel
TERM=xterm-color
```

This output comes from the bash shell on Mac OS X 10.3. Your mileage may vary, especially if you're running on Windows.

```
USER=niel
PERL5LIB=/sw/lib/perl5:/sw/lib/perl5
_=/usr/local/bin/mono
SGML_CATALOG_FILES=/sw/etc/sgml/catalog
PWD=/Users/niel/Documents/Mono Developers Notebook/monodn
SHELL=/bin/bash
SECURITYSESSIONID=210970
HOME=/Users/niel
PATH=/sw/bin:/sw/sbin:/bin:/sbin:/usr/bin:/usr/local/bin:/usr/sbin:/usr/
X11R6/bin:/usr/local/bin
SHLVL=1
```

How it works

There are four main interfaces in the `System.Collections` namespace that all collection types may choose to implement. They are: `ICollection`, the basic interface that indicates a class contains a collection of object instances; `IEnumerable`, which means that the class can be enumerated using the `foreach` statement; `IList`, which means the class contains an ordered list of object instances; and `IDictionary`, which means the class has a collection of keys and a collection of values.

The `System.Collections` namespace contains classes that can be used for the most common types of collections: `ArrayList`, an ordered list of objects; `BitArray`, a positional list of bits; `Hashtable`, a set of key-value pairs; `Queue`, a FIFO collection; `SortedList`, a collection of key-value pairs, kept ordered by keys; and `Stack`, a LIFO collection.

There's also the `System.Collections.Specialized` namespace, which contains more specialized collection types, like the `ListDictionary`, which is like a `Hashtable`, but is optimized for less than ten entries; and `HybridDictionary`, which uses a `ListDictionary` if the number of items is small, but switches to a `Hashtable` as the collection grows in size. There are also three collection types customized to work with strings rather than objects: `NameValueCollection`, a dictionary whose keys and values are strongly typed as strings; `StringCollection`, a collection of strings; and `StringDictionary`, a dictionary whose keys are strings but whose values may be any object.

Many namespaces contain other collection types, usually implementing one or more of the collection interfaces or extending one of the collection base types, such as `CollectionBase` or `DictionaryBase`. Quite often, you can deal with any concrete collection type as just an interface.

Example 3-5 deals with two of the collection interfaces, `IDictionary` and `ICollection`. `Environment.GetEnvironmentVariables()` returns an `IDictionary`; although it just happens to be a `Hashtable` under the

hood, you don't really need to know that. Similarly, the `IDictionary` interface defines a `Keys` property that returns an `ICollection`; in this case, the class under the hood is a private inner class of `Hashtable`, but it looks just like any other `ICollection` as far as we're concerned.

What about ...

...Creating your own collection types? The `System.Collections` namespace contains two handy classes that you can extend to build your own specialized collection types, `CollectionBase` and `DictionaryBase`. These two classes provide all the basic functionality needed to create your own specialized collection type.

For example, Example 3-6 shows the source code for a collection of `DateTime` objects. Descriptive comments are included in this listing.

Example 3-6. 04-collections/DateTimeCollection.cs

```
// 03-keyfunc/04-collections
using System;
using System.Collections;

public class DateTimeCollection : CollectionBase {

    // returns the object at the given index
    public DateTime this[int index] {
        get {
            return (DateTime)List[index];
        }
        set {
            List[index] = value;
        }
    }

    // adds the given object at the end of the collection
    public int Add(DateTime value) {
        return List.Add(value);
    }

    // returns the index of the object with the given value
    public int IndexOf(DateTime value) {
        return List.IndexOf(value);
    }

    // inserts the given value at the given index
    public void Insert(int index, DateTime value) {
        List.Insert(index, value);
    }

    // removes the object with the given value
    public void Remove(DateTime value) {
```

If you built your Mono installation with the right magical incantation, you'll have C# generics enabled. Generics provide typesafe collections at compile time, so you don't need to write the code to do it yourself. See "Use Generics" in Chapter 8 for more information on generics.

Example 3-6. 04-collections/DateTimeCollection.cs (continued)

```
List.Remove(value);
}

// returns true if the the collection contains an object
// with the given value
public bool Contains(DateTime value) {
    return List.Contains(value);
}

// called before the object is inserted
protected override void OnInsert(int index, object value) {
    Console.WriteLine("OnInsert({0},{1})", index, value);
    if (value.GetType() != typeof(DateTime)) {
        throw new ArgumentException(
            "value to be inserted must be of type System.DateTime",
            "value");
    }
}

// called before the object is removed
protected override void OnRemove(int index, object value) {
    Console.WriteLine("OnRemove({0},{1})", index, value);
    if (value.GetType() != typeof(DateTime)) {
        throw new ArgumentException(
            "value to be removed must be of type System.DateTime",
            "value");
    }
}

// called before the value at the given location is changed
protected override void OnSet(int index, object oldValue,
    object newValue) {
    Console.WriteLine("OnSet({0},{1},{2})", index, oldValue, newValue);
    if (newValue.GetType() != typeof(DateTime)) {
        throw new ArgumentException(
            "new value must be of type System.DateTime",
            "NewValue");
    }
}

// called when the object is validated
protected override void OnValidate(object value) {
    Console.WriteLine("OnValidate({0})", value);
    if (value.GetType() != typeof(DateTime)) {
        throw new ArgumentException(
            "value must be of type System.DateTime",
            "value");
    }
}

// test method
public static void Main(string [] args) {
```

Example 3-6. 04-collections/DateTimeCollection.cs (continued)

```
DateTimeCollection collection = new DateTimeCollection();
collection.Add(DateTime.Now);
collection.Add(new DateTime(2004,1,1));
// new instance of the same date/time value
collection.Remove(new DateTime(2004,1,1));
collection[0] = new DateTime(2001,12,31);
}
}
```

Voila! A collection class just for `DateTime` objects. Most of the members are fairly obvious, but the last four methods bear a little explanation.

The first three `On*()` methods are called before the value is actually inserted, removed, or set. By overriding them in the `DateTimeCollection`, you get a chance to check the value being passed before it is actually processed. The `OnValidate()` method is called before `OnInsert()`, `OnRemove()`, or `OnSet()`, so you get several chances to check the type of the object in question.

Compile and run *DateTimeCollection.exe* with these commands:

```
$ mcs DateTimeCollection.cs
$ mono DateTimeCollection.exe
OnValidate(Wednesday, 19 May 2004 00:21:08)
OnInsert(0,Wednesday, 19 May 2004 00:21:08)
OnValidate(Thursday, 01 January 2004 00:00:00)
OnInsert(1,Thursday, 01 January 2004 00:00:00)
OnValidate(Thursday, 01 January 2004 00:00:00)
OnRemove(1,Thursday, 01 January 2004 00:00:00)
OnValidate(Monday, 31 December 2001 00:00:00)
OnSet(0,Wednesday, 19 May 2004 00:21:08,Monday, 31 December 2001 00:00:00)
```

Work with Assemblies

As was already mentioned in “Package Related Classes with Assemblies” in Chapter 2, the assembly is the basic unit of deployment of a Mono library or application. In that lab, you saw how to create an assembly, how they are named, and how to store them in the GAC.

Everything's an assembly. What's in an assembly?

In this lab, you'll see how to examine the internal organization of assemblies.

How do I do that?

You can use the *monodis.exe* tool to view the CIL for any assembly. For example, the *SprocketSet.dll* assembly, which you'll remember from “Package Related Classes with Assemblies” in Chapter 2, is made up of

the classes `Sprocket` and `Widget`, along with attributes from `AssemblyInfo.cs` and resources from `SprocketSet.resources`. The contents of `Sprocket.cs` are:

```
public class Sprocket {  
}
```

and `Widget.cs` contains this:

```
public class Widget {  
    public Sprocket Sprocket;  
}
```

*Common
Intermediate
Language is the
equivalent of
assembly language
for the .NET
framework.*

Run `monodis SprocketSet.dll` to view the results in Example 3-7.

Example 3-7. 05-assemblies/SprocketSet.il

```
.assembly extern mscorlib  
{  
    .ver 1:0:5000:0  
}  
.assembly 'SprocketSet'  
{  
    .custom instance void class [mscorlib]'System.Reflection.  
AssemblyKeyNameAttribute'::.ctor(string) = ( 01 00 00 00 00 ) // .....  
  
    .custom instance void class [mscorlib]'System.Security.  
AllowPartiallyTrustedCallersAttribute'::.ctor() = ( 01 00 00 00 ) // ....  
  
    .custom instance void class [mscorlib]'System.Reflection.  
AssemblyTitleAttribute'::.ctor(string) = (  
    01 00 0B 53 70 72 6F 63 6B 65 74 53 65 74 00 00 ) // ...SprocketSet..  
  
    .custom instance void class [mscorlib]'System.Reflection.  
AssemblyDescriptionAttribute'::.ctor(string) = (  
    01 00 21 42 75 69 6C 64 20 53 70 72 6F 63 6B 65 // ...!Build Sprocke  
    74 73 20 66 6F 72 20 79 6F 75 72 20 77 69 64 67 // ts for your widg  
    65 74 73 2E 00 00 ) // ets..  
  
    .custom instance void class [mscorlib]'System.Reflection.  
AssemblyConfigurationAttribute'::.ctor(string) = ( 01 00 07 52 65 6C 65 61 73 65  
    00 00 ) // ...Release..  
  
    .custom instance void class [mscorlib]'System.Reflection.  
AssemblyCompanyAttribute'::.ctor(string) = (  
    01 00 13 41 6D 61 6C 67 61 6D 61 74 65 64 20 57 // ...Amalgamated W  
    69 64 67 65 74 73 00 00 ) // idgets..  
  
    .custom instance void class [mscorlib]'System.Reflection.  
AssemblyProductAttribute'::.ctor(string) = (  
    01 00 0B 53 70 72 6F 63 6B 65 74 53 65 74 00 00 ) // ...SprocketSet..  
  
    .custom instance void class [mscorlib]'System.Reflection.  
AssemblyCopyrightAttribute'::.ctor(string) = (  
    01 00 2E 32 30 30 34 20 41 6D 61 6C 67 61 6D 61 // ...2004 Amalgama
```

Example 3-7. 05-assemblies/SprocketSet.il (continued)

```
74 65 64 20 57 69 64 67 65 74 73 2E 20 41 6C 6C // ted Widgets. All
20 52 69 67 68 74 73 20 52 65 73 65 72 76 65 64 // Rights Reserved
2E 00 00 ) // ...

.custom instance void class [mscorlib]'System.Reflection.
AssemblyTrademarkAttribute'::.ctor(string) = (01 00 00 00 00 ) // .....

.custom instance void class [mscorlib]'System.Reflection.
AssemblyCultureAttribute'::.ctor(string) = (01 00 05 65 6E 2D 55 53 00 00 ) // .
..en-US..

.custom instance void class [mscorlib]'System.Reflection.
AssemblyDelaySignAttribute'::.ctor(bool) = (01 00 00 00 00 ) // .....

.custom instance void class [mscorlib]'System.Reflection.
AssemblyKeyFileAttribute'::.ctor(string) = (
01 00 0F 53 70 72 6F 63 6B 65 74 53 65 74 2E 73 // ...SprocketSet.s
6E 6B 00 00 ) // nk..

.custom instance void class [mscorlib]'System.CLSCompliantAttribute'::.
ctor(bool) = (01 00 01 00 00 ) // .....

.hash algorithm 0x00008004
.ver 1:0:1601:37419
.locale en-US
.publickey = (
00 24 00 00 04 80 00 00 94 00 00 00 06 02 00 00 // .$.....
00 24 00 00 52 53 41 31 00 04 00 00 11 00 00 00 // .$..RSA1.....
C3 9F ED 12 DB 31 A8 97 5A 3C D6 01 3F E6 09 22 // .....1..Z<...?"
76 F8 60 A0 A2 6D 49 DD 81 1C E5 12 FB 92 36 94 // v.`.mI.....6.
93 D8 EC 6D F8 3F D1 FC 4A 02 21 B7 6F 06 BE 18 // ...m.?.J.!o...
56 F0 7C 6B C0 D1 07 1A A8 8F 1E FD 38 5E A6 20 // V.|k.....8^
FD 36 86 E0 12 BE 91 89 DB C0 C2 D6 4F 5B FD 76 // .6.....0[.v
E1 47 16 8D 67 A0 E6 00 9E B3 A1 5B 75 09 8C 75 // .G..g.....[u..u
36 07 E8 31 E4 8B F2 6D 3B 12 28 0E 1C CC 75 45 // 6..1...m;(....uE
55 3B FD 55 BC 60 8E 7C 93 01 78 C6 3A 77 5E C6 ) // U;U.`.|.x.:w^
}
.module 'SprocketSet.dll' // GUID = {F7829D2A-5DAC-5B46-AD1D-D6CD2F7899CE}

.class public auto ansi beforefieldinit 'Sprocket'
extends [mscorlib]System.Object
{
// method line 1
.method public hidebysig specialname rtspecialname
instance default void .ctor ( ) cil managed
{
// Method begins at RVA 0x20ec
// Code size 7 (0x7)
.maxstack 8
IL_0000: ldarg.0
```

Example 3-7. 05-assemblies/SprocketSet.il (continued)

```
IL_0001: call instance void valuetype [mscorlib]'System.Object'::.ctor()
IL_0006: ret
    } // end of method Sprocket::instance default void .ctor ()

} // end of type Sprocket

.class public auto ansi beforefieldinit 'Widget'
    extends [mscorlib]System.Object
{
    .field public class 'Sprocket' 'Sprocket'

    // method line 2
    .method public hidebysig specialname rtspecialname
        instance default void .ctor () cil managed
    {
        // Method begins at RVA 0x20f4
        // Code size 7 (0x7)
        .maxstack 8
        IL_0000: ldarg.0
        IL_0001: call instance void valuetype [mscorlib]'System.Object'::.ctor()
        IL_0006: ret
    } // end of method Widget::instance default void .ctor ()

} // end of type Widget
```

How does it work?

Although there's not enough room in this notebook to explain this IL code in detail, you can see that it contains several sections. First, there's a reference to the external library *mscorlib.dll*, with the version number 1.0.3300.0:

```
.assembly extern mscorlib
{
    .ver 1:0:3300:0
}
```

You may notice that the order of the assembly attributes in the SprocketSet.il is not the same as in AssemblyInfo.cs. It doesn't matter; attributes are not order-sensitive.

Next, the assembly definition begins with these lines:

```
.assembly 'SprocketSet'
{
```

Following that, the assembly attributes begin. You'll recall that these attributes were listed in the file *AssemblyInfo.cs*.

```
.custom instance void class [mscorlib]'System.Reflection.
AssemblyDelaySignAttribute'::.ctor(bool) = (01 00 00 00 00 ) // .....
...
.custom instance void class [mscorlib]'System.CLSCompliantAttribute'::.
ctor(bool) = (01 00 01 00 00 ) // .....
```

Then the assembly info finishes up with this, which defines the hash algorithm for signing the assembly, and the version and locale of the assembly:

```
.hash algorithm 0x00008004  
.ver 1:0:1578:38317  
.locale en-US
```

That's the end of the assembly manifest. The next chunk of information is a *module*, which contains the actual classes for *SprocketSet.dll*. The module contains the classes *Sprocket* and *Widget*, including the IL for their data and method members.

If you still have questions about assemblies and IL after looking over Example 3-7, see *Inside Microsoft .NET IL Assembler* by Serge Lidin (Microsoft Press).

What about ...

...Decompiling a Mono assembly? *Decompiling* refers to the process of converting IL code back into the C# source that created it. There are two third-party tools that do just that: *Anakrino*, by Jay Freeman (<http://www.saurik.com/net/exemplar/>), and *Reflector*, by Lutz Roeder (<http://www.aisto.com/roeder/dotnet/>).

WARNING

If you decide to use a decompiler to look at Microsoft .NET assemblies, you may be asked *not* to contribute source code to the Mono development effort. The Mono team is very serious about maintaining a strict clean room in their development process.

Both of these tools require *System.Windows.Forms.dll* on your system, either by installing the Microsoft .NET Framework or by enabling *winelib* for Windows Forms compatibility.

Where to learn more

Look online at <http://msdn.microsoft.com/library/en-us/cpguide/html/cpconMicrosoftIntermediateLanguageMSIL.asp> to learn how to compile code to MSIL.

Start and Examine Processes

Often, a program needs to spawn other processes in order to get a job done. Unix programmers are familiar with the old `fork()` and `exec()` routine. Mono has equivalent functionality, and the same classes can be used to examine the state of other processes currently running.

In this lab, you'll see how to spawn processes, and how to look at the other processes currently running on your machine.

How do I do that?

Example 3-8 shows a program that runs the Unix `sleep` command, taking the number of seconds to sleep from a command-line argument.

Example 3-8. 06-process/StartProcess.cs

```
// 03-keyfunc/06-process
using System;
using System.Diagnostics;

public class StartProcess {
    public static void Main(string [] args) {
        string sleepTime = args[0];

        ProcessStartInfo startInfo = new ProcessStartInfo();
        startInfo.FileName = "sleep";
        startInfo.Arguments = sleepTime;
        Console.WriteLine("Starting {0} {1} at {2}",
            startInfo.FileName, startInfo.Arguments, DateTime.Now);
        Process process = Process.Start(startInfo);
        process.WaitForExit();
        Console.WriteLine("Done at {0}", DateTime.Now);
    }
}
```

Compile and run *StartProcess.cs* with the following command lines:

```
$ mcs StartProcess.cs
$ mono StartProcess.exe 10
Starting sleep 10 at Wednesday, 12 May 2004 21:24:47
Done at Wednesday, 12 May 2004 21:24:58
```

You can see that `sleep 10` was called, and that the start and end times were printed to standard output.

Example 3-9 contains another program using the `Process` class. This program, `ListProcesses`, lists all processes running on the machine, with their process ID, name, and start time.

Consciousness is a state of awareness of self and environment. Mono's got that.

Example 3-9. 06-process/ListProcesses.cs

```
using System;
using System.Diagnostics;

public class ListProcesses {
    public static void Main(string [] args) {
        foreach (Process process in Process.GetProcesses()) {
            Console.WriteLine("Process {0}: {1} on {2} started at {3}",
                process.Id, process.ProcessName, process.MachineName,
                process.StartTime);
        }
    }
}
```

When you compile and run *ListProcesses.exe*, you'll see something very much like this:

```
$ mcs ListProcesses.cs
$ mono ListProcesses.exe
Process 11687: bash started on localhost at Wednesday, 12 May 2004 21:11:19
Process 11691: bash started on localhost at Wednesday, 12 May 2004 21:12:27
Process 11806: ListProcesses started on localhost at Wednesday, 12 May 2004
21:13:43
```

How it works

The `System.Diagnostics` namespace contains a number of classes that can be used to provide information about the process environment currently in place. Among these is `Process`, which represents a single system process. The process can either be one that's currently running, one that you want to start, or one that has terminated but for which information still exists.

Example 3-8 shows a common technique for starting processes. The `ProcessStartInfo` class contains a number of properties that can be used to determine how a process is started, including `FileName`, the name of the file to be executed; `Arguments`, the argument list of the executable, as a single string; `WorkingDirectory`, the initial directory in which the process should be started; others that can be used to redirect standard input, output, and error; and others.

`Process` has a static method `Start()` that will take a `StartInfo` instance and start the corresponding process, returning a new `Process` instance.

Immediately after starting the process, you can call the `Process` class' `WaitForExit()` method to force the main Mono process to sleep until the new process terminates. This method has another overload that takes an `int` parameter, the maximum number of milliseconds to wait for the process to terminate.

Your results may be very different if you run on Windows. Unix and its brethren will only show you processes running under your user ID, while Windows will show every process running on the box.

Not all properties of the `Process` class will always contain data. It depends on how the process was started.

In Example 3-9, the `ListProcesses` class uses another method of `Process`. The static `GetProcesses()` method returns an array of `Processes`, each one containing information about a running process.

Multitask with Threads

Mono can walk and chew gum at the same time.

In addition to processes, as seen in “Start and Examine Processes,” Mono allows you to start and monitor threads within a single process.

In this lab, you’ll see how to start threads and ensure data synchronization between running threads.

How do I do that?

The thread pool is like a big bowl of noodles. When you’re hungry, you pull a noodle from the bowl and dip it in the sauce of your choice. Now make the noodle a Thread, the bowl the Thread-Pool, and the sauce a WaitCallback delegate.

Example 3-10 shows a program that uses the thread pool to queue a new work item that prints a line to the standard output at random intervals. The main thread continues to run until a carriage return is read from standard input.

Example 3-10. 07-threading/UseThreadPool.cs

```
// 03-keyfunc/07-threading
using System;
using System.IO;
using System.Threading;

public class UseThreadPool {
    private static Thread thread;

    public static void Main(string [] args) {

        WaitCallback callback = new WaitCallback(Callback);
        Console.WriteLine("Calling QueueUserWorkItem(...)");
        ThreadPool.QueueUserWorkItem(callback);

        Console.WriteLine("Hit return to exit.");
        Console.In.ReadLine();

        thread.Abort();

        Console.WriteLine("Done.");
    }

    private static void Callback(object state) {
        thread = Thread.CurrentThread;

        Console.WriteLine("Started thread {0}", thread.GetHashCode());

        Random random = new Random();
        for (int counter = 0; true; counter++) {
```

Example 3-10. 07-threading/UseThreadPool.cs (continued)

```
        try {
            Thread.Sleep(random.Next(10000));
        } catch (ThreadAbortException) {
            Console.WriteLine("Aborting thread");
        }
        Console.WriteLine("{0}: {1}", counter, DateTime.Now);
    }
}
```

Compiling and running *UseThreadPool.exe*, you'll see something like the following output:

```
$ mcs UseThreadPool.cs
$ mono UseThreadPool.exe
Calling QueueUserWorkItem()...
Hit return to exit.
Started thread 2069958092
0: Thursday, 06 May 2004 03:43:49
1: Thursday, 06 May 2004 03:43:52

Aborting thread
Done.
```

Note that you may see some of the output in a different order, depending on when the Thread is pulled from the ThreadPool and started.

How it works

Most of the time, it's recommended that when you need to start a new thread, you ask for a thread from the ThreadPool, as in Example 3-10.

The ThreadPool class provides access to a pool of Thread instances. By calling `QueueUserWorkItem`, you can request a Thread from the pool. When a Thread is available, it is started with the `WaitCallback` delegate that you specify, and the optional state object that you pass in. You don't have any control over when the Thread will become available, however, so this technique is best suited for times when immediate execution is not critical.

Regardless of how you start a thread, there may be times when you must ensure that only a single thread at a time has access to static data. Mono provides three good ways to do this, the `Monitor` class, the lock keyword, and the `System.Runtime.CompilerServices.MethodImpl` attribute. Under the hood they each do the same thing, but they provide variously more convenient ways of preventing threads from stepping on each other.

The `Monitor` class provides a way to explicitly synchronize threads. You can call the `Enter()` method, passing in an object to synchronize on, and if you later call the `Enter()` with the same synchronization object, that call will be blocked until the first thread calls `Leave()`. There are other methods to work with the thread lock in different ways, such as `TryEnter()` and `Wait()`.

The `lock` keyword provides the same functionality as `Monitor`, except that it automatically exits the monitor at the end of the block. For example, the following code block:

```
lock (someobject) {  
    // ...  
}
```

is exactly equivalent to this:

```
try {  
    Monitor.Enter(someobject);  
    // ...  
} finally {  
    Monitor.Exit(someobject);  
}
```

The final synchronization technique, the `MethodImpl` attribute, synchronizes a thread around an entire method call. So the following method definition:

```
using System.Runtime.CompilerServices;  
  
[MethodImpl(MethodImplOptions.Synchronized)]  
public void MyMethod() {  
    // ...  
}
```

compiles down to the following:

```
using System.Runtime.CompilerServices;  
  
public void MyMethod() {  
    try {  
        Monitor.Enter(this);  
        // ...  
    } finally {  
        Monitor.Exit(this);  
    }  
}
```

What about ...

...When you need more control over the Thread's execution? You can instantiate a `Thread`, passing in a `ThreadStart` delegate, and call its `Start()` method directly. Although you can be sure that the new thread

will start working immediately this way, the advantage may be mooted by the overhead of creating a new `Thread` instance. The thread pool keeps track of system utilization and will only create new `Thread` instances when absolutely necessary.

Example 3-11 shows how you can start a new thread, bypassing the thread pool.

Example 3-11. 07-threading/UseThreadStart.cs

```
// 03-keyfunc/07-threading
using System;
using System.IO;
using System.Threading;

public class UseThreadStart {
    private static Thread thread;

    public static void Main(string [] args) {

        ThreadStart start = new ThreadStart(Start);
        thread = new Thread(start);
        Console.WriteLine("Calling Start()...");
        thread.Start();

        Console.WriteLine("Hit return to exit.");
        Console.In.ReadLine();

        thread.Abort();

        Console.WriteLine("Done.");
    }

    private static void Start() {
        Console.WriteLine("Started thread {0}", thread.GetHashCode());

        Random random = new Random();
        for (int counter = 0; true; counter++) {
            try {
                Thread.Sleep(random.Next(10000));
            } catch (ThreadAbortException) {
                Console.WriteLine("Aborting thread");
            }
            Console.WriteLine("{0}: {1}", counter, DateTime.Now);
        }
    }
}
```

The differences between Example 3-10 and Example 3-11 are small, but important. To begin with, instead of creating a delegate of type `WaitCallback`, we create a `ThreadStart` delegate. Then, instead of calling `ThreadPool.QueueUserWorkItem()`, we instantiate a new `Thread`

object directly, and call its `Start()` method. Rather than waiting for a thread to be returned from the pool, the thread will start immediately.

Since we're creating a `Thread` instance directly in this program, there's no need for the `Start()` method to set the `thread` field in this case; we can just assign its value in the `Main()` method. Everything else remains the same.

Test Your C# Code

Does your code work? Does it really, really work?

It's a truism in software development that you can never find every bug. It also seems that just when you fix one bug, two more pop up to replace it. There are ways, however, to help prevent new errors from cropping up while working on new or existing code.

NUnit is a direct descendant of the JUnit Java unit testing framework, although it has taken on the characteristics of a C# framework as well. In this lab, we'll show you how you can use NUnit to test your code.

How do I do that?

Depending on how you installed Mono, you may or may not have NUnit on your system. If you find you do not have it already, you can install it from CVS module `mcs/nunit20` (see "Run a Development Version of Mono" in Chapter 8 for information on getting files from CVS). You may either copy `NUnit.Framework.dll` to your working directory, or install it in the GAC (see "Call External Libraries" in Chapter 2 for more information on using `gacutil.exe`).

Example 3-12 shows a test class called `MathTest`.

Example 3-12. 08-nunit/MathTest.cs

```
// 03-keyfunc/08-nunit
using System;

using NUnit.Framework;

[TestFixture]
public class MathTest {

    private int Zero = 0;
    private int One = 1;

    [SetUp]
    public void SetUp() {
        // do any set up the test requires
    }
}
```

Example 3-12. 08-nunit/MathTest.cs (continued)

```
[TearDown]
public void TearDown() {
    // clean up any data the test may have affected
}

[Test]
public void ZeroIs0() {
    Assertion.Assert("0 != Zero", 0 == Zero);
}

[Test]
public void OneIsNot0() {
    Assertion.Assert("0 == One", 0 != One);
}

[Test]
public void ZeroPlusZero() {
    Assertion.AssertEquals("0 + 0 != 0", Zero + Zero, Zero);
}

[Test]
[ExpectedException(typeof(DivideByZeroException))]
public void DivideByZero() {
    int i = One / Zero;
}

[Test]
public void NoObject() {
    object nullObject = null;
    Assertion.AssertNull("nullObject is not null", nullObject);
}

[Test]
public void ThisTestFails() {
    // this is designed to fail for demonstration purposes
    Assertion.Fail("This test failed");
}

[Test]
[Ignore("This test is not run at all")]
public void SkipThisTest() {
    // this test will be ignored for demonstration purposes
}
}
```

You'll notice that Example 3-12 has no `Main()` method. NUnit provides a test runner, called *nunit-console.exe*, so all you need to write is a suite of unit tests, not the infrastructure required to run them.

If you were testing another library, you would need to reference it with a `-r` argument as well.

There's also a shell script called `nunit-console` that wraps the `mono nunit-console.exe` command.

Again, JUnit users will remember that the `SetUp()` and `TearDown()` methods must have those exact names in JUnit. In NUnit, you can name them however you want, as long as they have the attribute indicating their usage.

Compile `MathTest.cs` just as you would any other Mono library. Use the following command line:

```
$ mcs -target:library -r:NUnit.Framework MathTest.cs
```

After compiling `MathTest.dll`, you can run the tests using `nunit-console.exe`. Use the following command line:

```
$ mono nunit-console.exe MathTest.dll
```

If all goes well, you'll see the following output:

```
NUnit version 1.0.5000
Copyright (C) 2002-2003 James W. Newkirk, Michael C. Two, Alexei A.
Vorontsov, Charlie Poole.
Copyright (C) 2000-2003 Philip Craig.
All Rights Reserved.

.....F.N
Tests run: 6, Failures: 1, Not run: 1, Time: 0.382437 seconds

Failures:
1) MathTest.ThisTestFails : This test failed
in <0x00020> MathTest.ThisTestFails ()
in (unmanaged) (wrapper managed-to-native) System.Reflection.MonoMethod:
InternalInvoke (object,object[])
in <0x0008c> (wrapper managed-to-native) System.Reflection.MonoMethod:
InternalInvoke (object,object[])
in <0x00104> System.Reflection.MonoMethod:Invoke (object,System.Reflection.
BindingFlags,System.Reflection.Binder,object[],System.Globalization.
CultureInfo)

Tests not run:
1) MathTest.SkipThisTest : This test is not run at all
```

How it works

Example 3-12 demonstrates how unit tests are written using NUnit. The classes from the NUnit framework are in the namespace `NUnit.Framework`, so the appropriate using statement is included in the code.

The first thing you'll notice is the `TestFixture` attribute. This attribute is used to indicate to the NUnit framework that the class `MathTest` is an NUnit *test fixture*; that is, it contains NUnit tests.

JUnit users will remember that in JUnit, test fixtures have to extend `TestCase` or implement `Test`. C# attributes make the same functionality possible without inheritance.

Each method within `MathTest` that needs to be visible to the NUnit framework is marked with at least one attribute. `SetUp`, for example, is used to indicate a method that will be called to set up any data for the test, and `TearDown` is used to indicate a method that will be called to clean up any leftover data from the test.

The tests themselves bear the `Test` attribute. The NUnit framework uses reflection to discover all the methods with the `SetUp`, `TearDown`, and `Test` attributes.

JUnit users know that the names of test methods have to start with `Test` for JUnit to discover them.

Two other attributes can be used to further define the test results. `ExpectedException` is useful for negative tests. It indicates that the method is expected to throw an exception, so that the exception indicates correct passage of the test.

The other attribute is `Ignore`. This attribute indicates that the test method should not be executed. The attribute has a single positional parameter, which is a string containing the reason why the test should not be run. This attribute is best used for test methods that you expect to fail temporarily, but you don't want the entire test session to show failures.

Within each test method, the `Assertion` class is used to find errors and report them as test failures. `Assertion` has a number of methods, with multiple overloads of each:

`Assert()`

Throws an `AssertionFailedError` exception if the value is false, optionally printing a specific message.

`AssertEquals()`

Throws an `AssertionFailedError` exception if two values are not equal, optionally printing a specific message.

`AssertNotNull()`

Throws an `AssertionFailedError` exception if the value is null, optionally printing a specific message.

`AssertNull()`

Throws an `AssertionFailedError` exception if the value is not null, optionally printing a specific message.

`AssertSame()`

Throws an `AssertionFailedError` exception if the values are not references to the same object, optionally printing a specific message.

`Fail()`

Causes the test to fail unconditionally, optionally printing a specific message.

When a test fixture is run with `nunit-console.exe`, the output indicates the tests passed, failed, and ignored by printing `.` for each test method found, `F` for a failed test, and `N` for a test not run, followed by a summary of the tests:

```
.....F.N
Tests run: 6, Failures: 1, Not run: 1, Time: 0.383369 seconds
```

Following the test summary information, specific information about any ignored and failed tests is printed, including any message passed into the assert method, and a stack trace (for failures):

```
Failures:
1) MathTest.ThisTestFails : This test failed
   in <0x00020> MathTest.ThisTestFails ()
   in (unmanaged) (wrapper managed-to-native) System.Reflection.MonoMethod:
   InternalInvoke (object,object[])
   in <0x0008c> (wrapper managed-to-native) System.Reflection.MonoMethod:
   InternalInvoke (object,object[])
   in <0x00104> System.Reflection.MonoMethod:Invoke (object,System.Reflection.
   BindingFlags,System.Reflection.Binder,object[],System.Globalization.
   CultureInfo)

Tests not run:
1) MathTest.SkipThisTest : This test is not run at all
```

What about ...

...Controlling the execution and output format of the NUnit test console? There are several ways to customize the output from *nunit-console.exe*. For example, if the test assembly contains more than one test fixture, you can specify which test fixture to run with the */fixture=classname* argument.

You can also control the output format by requesting output to an XML file with the */xml=filename* argument, optionally transforming it via an XSLT stylesheet specified in the */transform=stylesheet* argument; or you may output the XML to the console with the */xmlConsole* argument. If the standard text output is acceptable but you don't want the version banner printed, you can specify the */nologo* argument.

Where to learn more

The definitive reference for unit testing in all platforms and languages is <http://www.junit.org/>, the home of JUnit.

Documentation on NUnit, where it differs from JUnit, is available at <http://www.nunit.org/>.

Kent Beck's *Test Driven Development: By Example* (Addison-Wesley) is one of the seminal works in test driven development, and should be on the desk of every serious developer.

For a taste of how the Mono team uses NUnit, see the Mono Contributor Howto at <http://www.mono-project.com/contributing/testing.html>.