A Programmer's Introduction to C# 2.0

Third Edition

ERIC GUNNERSON AND NICK WIENHOLT

Apress®

A Programmer's Introduction to C# 2.0, Third Edition

ISBN (pbk): 1-59059-501-7

Printed and bound in the United States of America 9 8 7 6 5 4 3 2 1

Lead Editor: Jon Hassell
Technical Reviewer: Gavin Smyth
Editorial Board: Steve Anglin, Dan Appleman, Ewan Buckingham, Gary Cornell, Tony Davis, Jason Gilmore, Jonathan Hassell, Chris Mills, Dominic Shakeshaft, Jim Sumser
Assistant Publisher: Grace Wong
Project Manager: Kylie Johnston
Copy Edit Manager: Nicole LeClerc
Copy Editor: Kim Wimpsett
Production Manager: Kari Brooks-Copony
Compositors: Susan Glinert and Wordstop Technologies (P) Limited
Proofreader: Elizabeth Berry
Indexer: Broccoli Information Management
Artist: April Milne
Cover Designer: Kurt Krames
Interior Designer: Van Winkle Design Group
Manufacturing Manager: Tom Debolski

Distributed to the book trade in the United States by Springer-Verlag New York, Inc., 233 Spring Street, 6th Floor, New York, NY 10013, and outside the United States by Springer-Verlag GmbH & Co. KG, Tiergartenstr. 17, 69112 Heidelberg, Germany.

In the United States: phone 1-800-SPRINGER, fax 201-348-4505, e-mail `orders@springeronline.com`, or visit `http://www.springeronline.com`. Outside the United States: fax +49 6221 345229, e-mail `orders@springeronline.com`, or visit `http://www.springeronline.com`.

For information on translations, please contact Apress directly at 2560 Ninth Street, Suite 219, Berkeley, CA 94710. Phone 510-549-5930, fax 510-549-5939, e-mail `info@apress.com`, or visit `http://www.apress.com`.

The source code for this book is available to readers at `http://www.apress.com` in the Downloads section.

Contents at a Glance

Contents

Foreword to the Third Edition

"When do we get generics?"

Even before the first version of C# officially shipped, the C# language design team was getting feedback from customers. Overall, the feedback was positive, but developers were missing the flexibility of generic types (a facility also known as *templates* or *parameterized types* in some languages). This wasn't a surprise to the design team; we would have liked to have had generics in the original release as well, but doing them the right way—as a .NET runtime feature, accessible to all languages—was something that wouldn't fit into the schedule for v1.0, or even in the following v1.1 release.

Generic types are in the v2.0 version of C#, and with that addition, the top request of C# developers has been satisfied. Along with generics are three more new features also motivated by customer feedback: anonymous methods, iterators, and partial classes. While these additions don't have the scope or impact that generic types do, they're useful in specific situations. Combined with the new features in the C# 2.0 IDE, programming in C# has become much easier and more productive and is approaching the original vision of the designers.

Those of you who have earlier editions of this book may have noticed a new name on the cover. Because of a seemingly ever-growing list of commitments, I was unable to find enough time to extend the book to cover the new v2.0 features, so Nick lent his expertise to the majority of changes and additions in this edition. Together with the material from previous editions, I'm confident this edition contains the information you need to get inside the C# language and use it productively.

Stay sharp:
Eric Gunnerson
April 2005

Foreword to the First Two Editions

When you create a new programming language, the first question you're asked invariably is, why? In creating C#, we had several goals in mind:

To produce the first component-oriented language in the C/C++ family. Software engineering is less and less about building monolithic applications and more and more about building components that slot into various execution environments (for example, a control in a browser or a business object that executes in ASP+). Key to such components is that they have properties, methods, and events and that they have attributes that provide declarative information about the component. All of these concepts are first-class language constructs in C#, making it a very natural language in which to construct and use components.

To create a language in which everything really is an object. Through the innovative use of concepts such as boxing and unboxing, C# bridges the gap between primitive types and classes, allowing any piece of data to be treated as an object. Furthermore, C# introduces the concept of value types, which allows users to implement lightweight objects that don't require heap allocation.

To enable the construction of robust and durable software. C# was built from the ground up to include garbage collection, structured exception handling, and type safety. These concepts completely eliminate entire categories of bugs that often plague C++ programs.

To simplify C++ yet preserve the skills and investment programmers already have. C# maintains a high degree of similarity with C++, and programmers will immediately feel comfortable with the language. And C# provides great interoperability with COM and DLLs, allowing existing code to be fully leveraged.

We have worked very hard to attain these goals. A lot of the hard work took place in the C# design group, which met regularly over a period of two years. As head of the C# Quality Assurance team, Eric was a key member of the group, and through his participation he's eminently qualified to explain not only how C# works but also why it works that way. That will become evident as you read this book.

I hope you have as much fun using C# as those of us on the C# design team had creating it.

Anders Hejlsberg
Distinguished Engineer
Microsoft Corporation

About the Authors

After nearly a decade of programming at companies focusing on aerospace, databases, and bankruptcy, **ERIC GUNNERSON** was somewhat surprised to find himself working at Microsoft. He was the test lead for the Visual C++ compiler for several years, and then he became the test lead and joined the language design team for the language that was eventually named C#. After the first release of Visual C#, he experimented with the program manager role both on and off the language design team. He's currently a developer on the Windows Movie Maker team.

He blogs at `http://blogs.msdn.com/ericgu`, where he specializes in bad jokes, uninteresting and/or off-topic links, and the occasional nugget of C#-related content.

In his spare time, he enjoys skiing, cycling, home improvement, microcontroller-based holiday decorations, pinball, Halo 2, and writing about himself in the third person.

NICK WIENHOLT is an independent Windows and .NET consultant based in Sydney.

Nick has worked on a variety of IT projects over the past decade, ranging from numerical modeling of beach erosion to financial and payroll systems and from high-volume telecommunication number routing systems to digital rights management solutions for online movie providers. Nick specializes in system-level software architecture and development, with a particular focus on performance, security, interoperability, and debugging.

Nick is an active participant in the .NET community. He's the cofounder of the Sydney Deep .NET User Group; writes technical articles for *Australian Developer Journal*, ZDNet, Pinnacle Publishing, Developer.com, *MSDN Magazine* (Australia and New Zealand edition), and the Microsoft Developer Network; and is a keen participant in .NET-related newsgroups. An archive of Nick's SDNUG presentations, articles, and .NET blog is available at `http://www.dotnetperformance.com`.

In recognition of his work in the .NET area, he was awarded the Microsoft Most Valued Professional Award in 2002, 2003, and 2004.

Outside his professional interests, Nick is proud of his wonderful wife and daughter, is a keen Cronulla Sharks fan (and maintains a belief they will win a premiership in his lifetime), and loves reading technical books while lazing on Cronulla's fantastic beaches.

About the Technical Reviewer

GAVIN SMYTH is a professional software engineer with more years' experience in development than he cares to admit, ranging from device drivers to multihost applications, from real-time operating systems to Unix and Windows, from assembler to C++, and from Ada and C#. He has worked for clients such as Nortel, Microsoft, and BT, amongst others; he has written a few pieces as well (EXE and Wrox, where are you now?) but finds criticizing other people's work much more fulfilling. Beyond that, when he's not fighting weeds in the garden, he tries to persuade LEGO robots to do what he wants them to do (it's for the kids' benefit, honest).

Acknowledgments

Though writing a book is often a lonely undertaking, no author can do it without help.

I'd like to thank all those who helped me with the book, including all those team members who answered my incessant questions and read my unfinished drafts. I'd also like to thank my managers and Microsoft, both for allowing me to work on such a unique project and for allowing me to write a book about it.

Thanks to the Apress team for making a bet on an unproven author and for not pestering me when I waited to turn in content.

Thanks to all the artists who provided music to write to—all of which was commercially purchased—with special thanks to Rush for all their work.

Finally, I'd like to thank all those who supported me at home: my wife, Kim, and daughter, Samantha, who didn't complain when I was working, even when it was during our vacation, and my cat for holding my arms down while I was writing.

—Eric Gunnerson

Doing a book revision for a title that was one of your favorites is a strange and daunting exercise. Without the help and support of my editor and project manager—Jon and Kylie—this book would never have gotten done. We've had plenty of fun and interesting issues along the way, and I've certainly learned a lot about how different revising a book is from writing a new one.

As always, a big thanks to my family for their tolerance and support through yet another writing project.

—Nick Wienholt

Introduction

C# is one of the most exciting languages we've worked on and with. Most languages have strengths and weaknesses, but once in a while a new language comes along that meshes well with the hardware, software, and programming approaches of a specific time. We believe C# is such a language. Of course, language choice is often a "religious issue."[1]

We've structured this book as a tour through the language, since we think that's the best and most interesting way to learn a language. Unfortunately, tours can often be long and boring, especially if the material is familiar, and they sometimes concentrate on things you don't care about while overlooking things you're interested in. It's nice to be able to short-circuit the boring stuff and get into the interesting stuff. To do that, there are two approaches you might consider:

- To start things off quickly, skip to Chapter 3, which is a quick overview of the language and which gives enough information to start coding.
- To get a comparison of the language, skip to Chapter 42, which offers language-specific comparisons for C++, VB, and Java for programmers attuned to a specific language or for those who like to read comparisons.

After reading those chapters, you can then return to the beginning of the book or read each chapter in the order that interests you.

Why Another Language?

At this point, you're probably asking yourself, why should I learn another language? Why not use C++ (or VB or Java or whatever your preferred language is)? At least, you were probably asking yourself that before you bought the book.

Languages are a little bit like power tools. Each tool has its own strengths and weaknesses. Though we *could* use a router to trim a board to length, it'd be much easier to use a miter saw. Similarly, we could use a language such as LISP to write a graphics-intensive game, but it'd probably be easier to use C++.

C# (pronounced "C sharp") is the native language for the .NET common language runtime (CLR). It has been designed to fit seamlessly into the .NET CLR. You can (and, at times, you should) write code in either Visual C++ or Visual Basic, but in most cases, C# will likely fit your needs better. Because the CLR is central to many things in C#, Chapter 2 introduces the important parts of it—at least, those that are important to the C# language.

1. See the Jargon File (`http://www.jargonfile.org`) for a good definition of *religious issue*.

C# Design Goals

When the C++ language first came out, it caused quite a stir. Here was a language for creating object-oriented software that didn't require C programmers to abandon their skills or their investment in software. It wasn't fully object-oriented in the way a language like Eiffel is, but it had enough object-oriented features to offer great benefits.

C# provides a similar opportunity. In cooperation with the .NET CLR, it provides a language to use for component-oriented software, without forcing programmers to abandon their investment in C, C++, or COM code.

C# is designed for building robust and durable components to handle real-world situations.

Component Software

The .NET CLR is a component-based environment, and it should come as no surprise that C# is designed to make component creation easier. It's a "component-centric" language, in that all objects are written as components, and the component is the center of the action.

Component concepts, such as properties, methods, and events, are first-class citizens of the language and of the underlying runtime environment. Declarative information (known as *attributes*) can be applied to components to convey design-time and runtime information about the component to other parts of the system. Documentation can be written inside the component and exported to XML.

C# objects don't require header files, IDL files, or type libraries to be created or used. Components created by C# are fully self-describing and can be used without a registration process.

C# is aided in the creation of components by the .NET runtime and .NET Framework, which provide a unified type system in which everything can be treated as an object but without the performance penalty associated with pure object systems, such as Smalltalk.

Robust and Durable Software

In the component-based world, being able to create software that's robust and durable is important. Web servers may run for months without a scheduled reboot, and an unscheduled reboot is undesirable.

Garbage collection takes the burden of memory management away from the programmer,[2] and the problems of writing versionable components are eased by definable versioning semantics and the ability to separate the interface from the implementation. Numerical operations can be checked to ensure that they don't overflow, and arrays support bounds checking.

C# also provides an environment that's simple, safe, and straightforward. Error handling isn't an afterthought, with exception handling being present throughout the environment. The language is type-safe, and it protects against the use of variables that have not been initialized, unsafe casts, and other common programming errors.

---

2. It's not that C++ memory management is conceptually hard—it isn't in most cases, but there are some difficult situations when dealing with components. The burden comes from having to devote time and effort to getting it right. With garbage collection, it isn't necessary to spend the coding and testing time to make sure there aren't any memory leaks, which frees the programmer to focus on the program logic.

Real-World Software

Software development isn't pretty. Software is rarely designed on a clean slate; it must have decent performance, leverage existing code, and be practical to write in terms of time and budget. A well-designed environment is of little use if it doesn't provide enough power for real-world use.

C# provides the benefits of an elegant and unified environment while still providing access to "less reputable" features—such as pointers—when those features are needed to get the job done.

C# protects the investment in existing code. Existing COM objects can be used as if they were .NET objects.[3] The .NET CLR will make objects in the runtime appear to be COM objects to existing COM-based code. Native C code in DLL files can be called from C# code.[4]

C# provides low-level access when appropriate. Lightweight objects can be written to be stack allocated and still participate in the unified environment. Low-level access is provided via the `unsafe` mode, which allows pointers to be used in cases where performance is important or when pointers are required to use existing DLLs.

C# is built on a C++ heritage and should be immediately comfortable for C++ programmers. The language provides a short learning curve, increased productivity, and no unnecessary sacrifices.

Finally, C# capitalizes on the power of the .NET CLR, which provides extensive library support for general programming tasks and application-specific tasks. The .NET runtime, the .NET Framework, and the .NET languages are all tied together by the Visual Studio environment, providing one-stop shopping for the .NET programmer.

Second Edition Updates

Compared with the first edition, the second edition of this book included updates of all the samples to conform to the compiler's beta 2 release. Most of these changes were fairly minor, mainly based on naming changes in the frameworks, though some of the samples did require a bit of rearchitecting.

The second set of changes typically involved the addition of small sections or new examples.

As for the major changes, the second edition contained heavily revised chapters on delegates and events and showed how to develop a sample application using Windows Forms. The book contained a new chapter on threading and asynchronous operations, which detailed two ways of getting things to occur simultaneously. Finally, it included a new chapter on execution-time code generation, which detailed how to write a self-modifying application.

Third Edition Updates

As you can well imagine, a lot has changed since the second edition of this book. Yet again, the code is based on a beta 2 release, but this time the beta is of .NET 2.0. Since the second edition, .NET and C# have been released and received widespread support and adoption. This adoption has been so successful that the recent development survey conducted by Computerworld now

---

3. Usually. Certain details sometimes make this a bit tougher in practice.
4. For C++ code, Visual C++ has been extended with Managed Extensions that make it possible to create .NET components. In .NET 2.0, Managed Extensions have been replaced by C++/CLI, which offers a syntax similar to C#.

ranks C# as the world's most popular language (`http://www.computerworld.com/developmenttopics/development/story/0,10801,100542,00.html`).

The C# language and the .NET Framework have undergone significant changes in the move from .NET 1.*x* to .NET 2.0, and this book has been extensively updated to reflect these changes. In places where a .NET 1.*x* feature has been superseded by a .NET 2.0 feature, the coverage of the original material has generally been added to rather than removed. The main motivation for this is to support developers who need to have their code work on any version of the .NET Framework. Updating clients to newer versions of the framework can be a significant undertaking, and in many scenarios, a code base will need to work on all released versions of the framework, which excludes the use of C# 2.0 features.

In addition to the new chapters on generics and nullable types, we've added the following material to existing chapters:

- Static classes
- Partial classes
- Assembly aliases
- Secure strings
- Property accessor accessibility modifiers
- Iterators
- Anonymous methods
- Inline warning control
- Generic collection classes
- Semaphores
- Lightweight code generation
- Fixed-size buffers
- ClickOnce deployment
- Updated information on the garbage collector
- New compiler switches

C# continues to be at the forefront of language innovation on the .NET platform, and this edition covers all the new features that will allow you to stay at the cutting-edge of software development.

The C# Compiler and Other Resources

You have two ways of getting the C# compiler. The first is as part of the .NET SDK.

The SDK contains compilers for C#, VB, C++, and all of the frameworks. After you install the SDK, you can compile C# programs using the `csc` command, which will generate an EXE that you can execute.

The other way of getting the compiler is as part of Visual Studio .NET. The beta of Visual Studio .NET 2005 is currently available, with the final release scheduled for late 2005.

To find out more about getting the .NET SDK or the Visual Studio .NET 2005 beta, please consult this book's page on the Apress Web site at `http://www.apress.com`.

Compiler Hints

When compiling code, the C# compiler must be able to locate information about the components that are being used. It will automatically search the file named `mscorlib.dll`, which contains the lowest-level .NET entities, such as data types.

To use other components, the appropriate DLL for that component must be specified on the command line. For example, to use WinForms, you must specify the `system.winforms.dll` file as follows:

```
csc /r:system.winforms.dll myfile.cs
```

The usual naming convention is for the DLL to be the same as the namespace name.

Other Resources

Microsoft maintains public newsgroups for .NET programming. The C# newsgroup is named `microsoft.public.dotnet.csharp.general`, and it lives on the `msnews.microsoft.com` news server.

Numerous Web sites are devoted to .NET information. You can find links to these resources at the Apress Web site.

CHAPTER 1

Object-Oriented Basics

This chapter introduces object-oriented programming. Those who are familiar with object-oriented programming will probably want to skip this chapter.

You can take many approaches to object-oriented design, as evidenced by the number of books written about it. The following introduction takes a fairly pragmatic approach and doesn't spend a lot of time on design, but the design-oriented approaches can be quite useful to newcomers.

What's an Object?

An *object* is merely a collection of related information and functionality. An object can be something that has a corresponding real-world manifestation (such as an `employee` object), something that has some virtual meaning (such as a window on the screen), or just some convenient abstraction within a program (a list of work to be done, for example).

An object contains the data that describes the object and the operations that can be performed on the object. Information stored in an `employee` object, for example, might be various identification information (name and address), work information (job title and salary), and so on. The operations performed might include creating an employee paycheck or promoting an employee.

When creating an object-oriented design, the first step is to determine what the objects are. When dealing with real-life objects, this is often straightforward, but when dealing with the virtual world, the boundaries become less clear. That's where the art of good design shows up, and it's why good architects are in such demand.

Inheritance

Inheritance is a fundamental feature of an object-oriented system, and it's simply the ability to inherit data and functionality from a parent object. Rather than developing new objects from scratch, new code can be based on the work of other programmers,[1] adding only the new features that are needed. The parent object that the new work is based upon is known as a *base class*, and the child object is known as a *derived class*.

Inheritance gets a lot of attention in explanations of object-oriented design, but the use of inheritance isn't particularly widespread in most designs. There are several reasons for this.

1. At this point perhaps we should say something about "standing on the shoulders of giants…."

First, inheritance is an example of what's known in object-oriented design as an "is-a" relationship. If a system has an `animal` object and a `cat` object, the `cat` object could inherit from the `animal` object because a `cat` "is-a" `animal`. In inheritance, the base class is always more generalized than the derived class. The `cat` class would inherit the `eat` function from the animal class and would have an enhanced `sleep` function. In real-world design, such relationships aren't particularly common.

Second, to use inheritance, the base class needs to be designed with inheritance in mind. This is important for several reasons. If the objects don't have the proper structure, inheritance can't really work well. More important, a design that enables inheritance also makes it clear that the author of the base class is willing to support other classes inheriting from the class. If a new class is inherited from a class where this isn't the case, the base class might at some point change, breaking the derived class.

Some less-experienced programmers mistakenly believe that inheritance is "supposed to be" used widely in object-oriented programming and therefore use it far too often. Inheritance should be used only when the advantages it brings are needed.[2] See the upcoming "Polymorphism and Virtual Functions" section.

In the .NET common language runtime (CLR), all objects are inherited from the ultimate base class named `object`, and there's only single inheritance of objects (in other words, an object can be derived from only one base class). This prevents the use of some common idioms available in multiple-inheritance systems such as C++, but it also removes many abuses of multiple inheritance and provides a fair amount of simplification. In most cases, it's a good trade-off. The .NET runtime allows multiple inheritance in the form of interfaces, which can't contain implementation. We'll discuss interfaces in Chapter 10.

Containment

So, if inheritance isn't the right choice, what is?

The answer is *containment*, also known as *aggregation*. Rather than saying an object is an example of another object, an instance of that other object will be contained inside the object. So, instead of having a class look like a string, the class will contain a string (or an array or a hash table).

The default design choice should be containment, and you should switch to inheritance only if needed (in other words, if there really is an "is-a" relationship).

Polymorphism and Virtual Functions

Once, while writing a music system, we decided we wanted to be able to support both WinAmp and Windows Media Player as playback engines, but we didn't want all the code to have to know which engine it was using. We therefore defined an *abstract class*, which is a class that defines the functions a derived class must implement and that sometimes provides functions that are useful to both classes.

2. Perhaps someone should write a paper called "Multiple Inheritance Considered Harmful." Someone, someplace probably has….

In this case, the abstract class was called `MusicServer`, and it had functions such as `Play()`, `NextSong()`, `Pause()`, and so on. Each of these functions was declared as abstract so each player class would have to implement those functions themselves.

Abstract functions are automatically *virtual functions*, which allow the programmer to use polymorphism to make their code simpler. When there's a virtual function, the programmer can pass around a reference to the abstract class rather than the derived class, and the compiler will write code to call the appropriate version of the function at runtime.

An example will probably make this clearer. The music system supports both WinAmp and Windows Media Player as playback engines. The following is a basic outline of what the classes look like:

```
using System;
public abstract class MusicServer
{
    public abstract void Play();
}
public class WinAmpServer: MusicServer
{
    public override void Play()
    {
        Console.WriteLine("WinAmpServer.Play()");
    }
}
public class MediaServer: MusicServer
{
    public override void Play()
    {
        Console.WriteLine("MediaServer.Play()");
    }
}
class Test
{
    public static void CallPlay(MusicServer ms)
    {
        ms.Play();
    }
    public static void Main()
    {
        MusicServer ms = new WinAmpServer();
        CallPlay(ms);
        ms = new MediaServer();
        CallPlay(ms);
    }
}
```

This code produces the following output:

```
WinAmpServer.Play()
MediaServer.Play()
```

Polymorphism and virtual functions are used in many places in the .NET runtime system. For example, the base object `object` has a virtual function called `ToString()` that's used to convert an object into a string representation of the object. If you call the `ToString()` function on an object that doesn't have its own version of `ToString()`, the version of the `ToString()` function that's part of the `object` class will be called,[3] which simply returns the name of the class. If you *overload*—write your own version of—the `ToString()` function, that one will be called instead, and you can do something more meaningful, such as writing out the name of the employee contained in the `employee` object. In the music system, this meant overloading functions for play, pause, next song, and so on.

Encapsulation and Visibility

When designing objects, the programmer gets to decide how much of the object is visible to the user and how much is private within the object. Details that aren't visible to the user are said to be *encapsulated* in the class.

In general, the goal when designing an object is to encapsulate as much of the class as possible. These are the most important reasons for doing this:

- The user can't change private things in the object, which reduces the chance the user will either change or depend upon such details in their code. If the user does depend on these details, changes made to the object may break the user's code.
- Changes made in the public parts of an object must remain compatible with the previous version. The more that's visible to the user, the fewer things that can be changed without breaking the user's code.
- Larger interfaces increase the complexity of the entire system. Private fields can be accessed only from within the class; public fields can be accessed through any instance of the class. Having more public fields often makes debugging much tougher.

Chapter 5 will explore this subject further.

3. Or, if there's a base class of the current object and it defines `ToString()`, that version will be called.

CHAPTER 2

The .NET Runtime Environment

In the past, writing modules that could be called from multiple languages was difficult. Code that's written in Visual Basic can't be called from Visual C++. Code that's written in Visual C++ can sometimes be called from Visual Basic, but it's not easy to do. Visual C++ uses the C and C++ runtimes, which have specific behavior, and Visual Basic uses its own execution engine, also with its own specific—and different—behavior.

And so the Component Object Model (COM) was created, and it has been pretty successful as a way of writing component-based software. Unfortunately, it's fairly difficult to use from the Visual C++ world, and it's not fully featured in the Visual Basic world. And therefore, it was used extensively when writing COM components but was used less often when writing native applications. So, if one programmer wrote some nice code in C++ and another wrote some in Visual Basic, there really wasn't an easy way to work together.

Further, the world was tough for library providers, as no one choice would work in all markets. If the writer thought the library was targeted toward the Visual Basic crowd, it'd be easy to use from Visual Basic, but that choice might either constrain access from the C++ perspective or come with an unacceptable performance penalty. Or, a library could be written for C++ users for good performance and low-level access, but it'd ignore the Visual Basic programmers.

Sometimes a library would be written for both types of users, but this usually meant some compromises had to happen. To send e-mail on a Windows system, for example, you have a choice between Collaboration Data Objects (CDO), which is a COM-based interface that can be called from both languages but doesn't do everything,[1] and native Messaging Application Programming Interface (MAPI) functions (in both C and C++ versions) that can access all functions.

The .NET runtime is designed to remedy this situation. It has one way of describing code (metadata) and one runtime and library (the CLR and .NET Framework). Figure 2-1 shows how the .NET runtime is arranged.

The CLR provides the basic execution services. On top of that, the base classes provide basic data types, collection classes, and other general classes. Built on top of the base classes are classes for dealing with data and Extensible Markup Language (XML). Finally, at the top of the architecture are classes to expose Web services[2] and to deal with the user interface. An application may call in at any level and use classes from any level.

1. Presumably this is because it's difficult to translate the low-level internal design into something that can be called from an automation interface.
2. This is a way to expose a programmatic interface via a Web server.

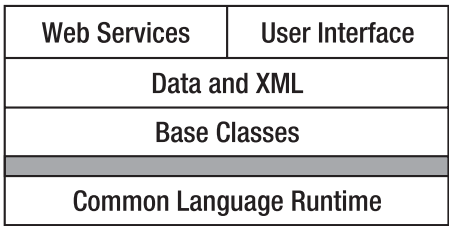

Figure 2-1. *.NET Framework organization*

To understand how C# works, it's important to understand a bit about the .NET runtime and the .NET Framework. The following section provides an overview; you can find more detailed information in Chapter 38.

The Execution Environment

This section was once titled "The Execution Engine," but the .NET runtime is much more than just an engine. The environment provides a simpler programming model, safety and security, powerful tools support, and help with deployment, packaging, and other support.

A Simpler Programming Model

All services are offered through a common model that can be accessed equally through all the .NET languages, and the services can be written in any .NET language.[3] The environment is largely language-agnostic, allowing language choice. This makes code reuse easier, both for the programmer and for the library providers.

The environment also supports using existing code in C# code, either through calling functions in Dynamic Link Libraries (DLLs) or making COM components appear to be .NET runtime components. .NET runtime components can also be used in situations that require COM components.

In contrast with the various error-handling techniques in existing libraries, in the .NET runtime all errors are reported via exceptions. You have no need to switch between error codes, `HRESULT`s, and exceptions.

Finally, the environment contains the .NET Framework, which provides the functions traditionally found in runtime libraries, plus a few new ones. The framework is divided into different categories.

System

The `System` namespace contains the core classes for the runtime. These classes are roughly analogous to the C++ runtime library and include the nested namespaces described in Table 2-1.

3. Some languages may not be able to interface with native platform capabilities.

Table 2-1. *System Namespace*

Namespace	Function
Collections	Contains collection objects, such as lists, queues, and hash tables
Configuration	Contains configuration and installation objects
Diagnostics	Debugs and traces execution of code
Globalization	Globalizes your application
IO	Performs input and output
Net	Performs network operations
Reflection	Views the metadata of types and dynamically loads and creates objects
Security	Supports the .NET security system
ServiceProcess	Creates and manages Windows services
Text	Contains encoding and conversion classes
Threading	Contains threads and synchronization
Runtime	Contains interop, remoting, and serialization

System.Data

The System.Data namespace contains the classes that support database operations (see Table 2-2).

Table 2-2. *System.Data Namespace*

Section	Function
ADO	ADO.NET
Design	Design-time database support
SQL	SQL Server support
SQLTypes	Data types for SQL server

System.Xml

The System.Xml namespace contains classes to manage XML.

System.Drawing

The System.Drawing namespace contains classes that support GDI+, including printing and imaging.

System.Web

The `System.Web` namespace contains classes for dealing with Web services and classes for creating Web-based interfaces using ASP.NET.

System.Windows.Forms

The `System.Windows.Forms` namespace contains classes to create rich-client interfaces.

Safety and Security

The .NET runtime environment is designed to be a safe and secure environment. The .NET runtime is a managed environment, which means that the runtime manages memory for the programmer. Instead of having to manage memory allocation and deallocation, the garbage collector does it. Not only does garbage collection reduce the number of things to remember when programming, in a server environment it can drastically reduce the number of memory leaks. This makes high-availability systems much easier to develop.

Additionally, the .NET runtime is a verified environment. At runtime, the environment verifies that the executing code is type-safe. This can catch errors, such as passing the wrong type to a function, and can catch attacks, such as trying to read beyond allocated boundaries or executing code at an arbitrary location.

The security system interacts with the verifier to ensure that code does only what it's permitted to do. The security requirements for a specific piece of code can be expressed in a finely grained manner; code can, for example, specify that it needs to be able to write a scratch file, and that requirement will be checked during execution.

Powerful Tools Support

Microsoft supplies four .NET languages: Visual Basic, C#, C++/CLI and J#. Other companies are working on compilers for other languages that run the gamut from COBOL to Perl.

Debugging is greatly enhanced in the .NET runtime. The common execution model makes cross-language debugging simple and straightforward, and debugging can seamlessly span code written in different languages and running in different processes or on different machines.

Finally, all .NET programming tasks are tied together by the Visual Studio environment, which gives support for designing, developing, debugging, and deploying applications.

Deployment, Packaging, and Support

The .NET runtime helps out in these areas as well. Deployment has been simplified, and in some cases there isn't a traditional install step. Because the packages are deployed in a general format, a single package can run in any environment that supports .NET. Finally, the environment separates application components so that an application runs only with the components it shipped with, rather than with different versions shipped by other applications.

.NET 2.0 again simplifies the deployment process with a new technology called ClickOnce, which allows a Windows Forms application to be deployed in a manner that's conceptually similar to the deployment model of a Web application. Chapter 37 walks you through deploying an application with ClickOnce.

Metadata

Metadata is the glue that holds the .NET runtime together. Metadata is the analog of the type library in the COM world but with much more extensive information.

For every object that's part of the .NET world, the metadata for that object records all the information that's required to use the object, which includes the following:

- The name of the object
- The names of all the fields of the object and their types
- The names of all member functions, including parameter types and names

With this information, the .NET runtime is able to figure out how to create objects, call member functions, or access object data, and compilers can use them to find out what objects are available and how an object is used.

This unification is nice for both the producer and the consumer of code; the producer of code can easily author code that can be used from all .NET-compatible languages, and the user of the code can easily use objects created by others, regardless of the language that the objects are implemented in.

Additionally, this rich metadata allows other tools access to detailed information about the code. The Visual Studio shell uses this information in the Object Browser and for features such as IntelliSense.

Finally, runtime code can query the metadata—in a process called *reflection*—to find out what objects are available and what functions and fields are present on the class. This is similar to dealing with IDispatch in the COM world but with a simpler model. Of course, such access isn't strongly typed, so most software will choose to reference the metadata at compile time rather than runtime, but it's a useful facility for applications such as scripting languages.

Finally, reflection is available to the end user to determine what objects look like, to search for attributes, or to execute methods whose names aren't known until runtime.

Assemblies

In the past, a finished software package might have been released as an executable, as DLL and LIB files, as a DLL containing a COM object and a typelib, or as some other mechanism.

In the .NET runtime, the mechanism of packaging is the *assembly*. When code is compiled by one of the .NET compilers, it's converted to an intermediate form known as Intermediate Language (IL). The assembly contains all the IL, metadata, and other files required for a package to run—in one complete package. Each assembly contains a manifest that enumerates the files contained in the assembly, controls what types and resources are exposed outside the assembly, and maps references from those types and resources to the files that contain the types and resources. The manifest also lists the other assemblies that an assembly depends upon.

Assemblies are self-contained; enough information exists in the assembly for it to be self-describing.

When defining an assembly, the assembly can be contained in a single file, or it can be split amongst several files. Using several files will enable a scenario where sections of the assembly are downloaded only as needed.

Language Interop

One of the goals of the .NET runtime is to be language-agnostic, allowing code to be used and written from whatever language is convenient. Not only can classes written in Visual Basic be called from C# or C++ (or any other .NET language), a class that was written in Visual Basic can be used as a base class for a class written in C#, and that class could be used from a C++ class.

In other words, it shouldn't matter which language a class was authored in. In fact, it often isn't possible to tell what language a class was written in.

In practice, this goal runs into a few obstacles. Some languages have unsigned types that aren't supported by other languages, and some languages support operator overloading. Allowing the more feature-rich languages to retain their freedom of expression while still making sure their classes can interop with other languages is challenging.

To support this, the .NET runtime has sufficient support to allow the feature-rich languages full expressibility,[4] so code that's written in one of those languages isn't constrained by the simpler languages.

For classes to be usable from .NET languages in general, the classes must adhere to the Common Language Specification (CLS), which describes what features can be visible in the public interface of the class (any features can be used internally in a class). For example, the CLS prohibits exposing unsigned data types because not all languages can use them. You can find more information on the CLS in the "Cross-Language Interoperability" section of the .NET software development kit (SDK).

A user writing C# code can indicate that it's supposed to be CLS compliant, and the compiler will flag any noncompliant areas. For more information on the specific restrictions placed on C# code by CLS compliance, see Chapter 38.

Attributes

To transform a class into a component, you'll often need some additional information, such as how to persist a class to disk or how transactions should be handled. The traditional approach is to write the information in a separate file and then combine it with the source code to create a component.

The problem with this approach is that information is duplicated in multiple places. It's cumbersome and error-prone, and it means you don't have the whole component unless you have both files.[5]

The .NET runtime supports custom attributes (known simply as *attributes* in C#), which are a way to place descriptive information in the metadata along with an object and then retrieve the data later. Attributes provide a general mechanism for doing this, and they're used heavily throughout the runtime to store information that modifies how the runtime uses the class.

Attributes are fully extensible, and this allows programmers to define attributes and use them.

4. This isn't quite true for C++/CLI, which loses some expressibility over C++.
5. Anybody who has ever tried to do COM programming without a typelib should understand the problem with this.

CHAPTER 3

C# Quick Start and C# Development

This chapter presents a quick overview of the C# language. This chapter assumes a certain level of programming knowledge and therefore doesn't present very much detail. If the explanation here doesn't make sense, look for a more detailed explanation of the particular topic later in the book.

The second part of the chapter discusses how to obtain the C# compiler and the advantages of using Visual Studio .NET (VS .NET) to develop C# applications.

Hello, Universe

As a supporter of SETI,[1] we thought it'd be appropriate to do a "Hello, Universe" program rather than the canonical "Hello, World" program:

```
using System;
class Hello
{
    public static void Main(string[] args)
    {
        Console.WriteLine("Hello, Universe");

            // iterate over command-line arguments,
            // and print them out
        for (int arg = 0; arg < args.Length; arg++)
            Console.WriteLine("Arg {0}: {1}", arg, args[arg]);
    }
}
```

As discussed earlier, the .NET runtime has a unified namespace for all program information (or metadata). The `using System` clause is a way of referencing the classes that are in the `System` namespace so they can be used without having to put `System` in front of the type name.

1. Search for Extraterrestrial Intelligence. See `http://www.teamseti.org` for more information.

The `System` namespace contains many useful classes, one of which is the `Console` class, which is used (not surprisingly) to communicate with the console (or DOS box or command line, for those who have never seen a console).

Because C# doesn't have global functions, the example declares a class called `Hello` that contains the static `Main()` function, which serves as the starting point for execution. `Main()` can be declared with no parameters or with a string array. Since it's the starting function, it must be a static function, which means it isn't associated with an instance of an object.

The first line of the function calls the `WriteLine()` function of the `Console` class, which will write "Hello, Universe" to the console. The `for` loop iterates over the parameters that are passed in and then writes out a line for each parameter on the command line.

Namespaces and using

Namespaces in the .NET runtime organize classes and other types into a single hierarchical structure. The proper use of namespaces will make classes easy to use and prevent collisions with classes written by other authors.

You can also think of namespaces as a way to specify really long names for classes and other types without having to always type a full name.

Namespaces are defined using the `namespace` statement. For multiple levels of organization, namespaces can be nested:

```
namespace Outer
{
    namespace Inner
    {
        class MyClass
        {
            public static void Function() {}
        }
    }
}
```

That's a fair amount of typing and indenting, so it can be simplified by using the following instead:

```
namespace Outer.Inner
{
    class MyClass
    {
        public static void Function() {}
    }
}
```

Each source file can define as many different namespaces as needed.

As mentioned in the "Hello, Universe" section, `using` imports the metadata for types into the current program so the types can be more easily referenced. The `using` keyword is merely a shortcut that reduces the amount of typing that's required when referring to elements, as Table 3-1 indicates.

Table 3-1. *Code Differences with the* `using` *Keyword*

`using` Clause	Source Line
<none>	`System.Console.WriteLine("Hello");`
`using System`	`Console.WriteLine("Hello");`

Collisions between types or namespaces that have the same name can always be resolved by a type's fully qualified name. This could be a long name if the class is deeply nested, so the following is a variant of the `using` clause that allows an alias to be defined to a class:

```
using ThatConsoleClass = System.Console;
class Hello
{
    public static void Main()
    {
        ThatConsoleClass.WriteLine("Hello");
    }
}
```

To make the code more readable, the examples in this book rarely use namespaces, but you should use them in most real code.

Namespaces and Assemblies

An object can be used from within a C# source file only if the C# compiler can locate that object. By default, the compiler will open only the single assembly known as `mscorlib.dll`, which contains the core functions for the CLR.

To reference objects located in other assemblies, the name of the assembly file must be passed to the compiler. You can do this on the command line using the `/r:<assembly>` option or from within Visual Studio by adding a reference to the C# project.

Typically, a correlation exists between the namespace an object is in and the name of the assembly in which it resides. For example, the types in the `System.Net` namespace reside in the `System.Net.dll` assembly. Types are usually placed in assemblies based on the usage patterns of the objects in that assembly; a large or rarely used type in a namespace might be placed in its own assembly.

You can find the exact name of the assembly that an object is contained in within the documentation for that object.

Basic Data Types

C# supports the usual set of data types. For each data type that C# supports, there's a corresponding underlying .NET CLR type. For example, the `int` type in C# maps to the `System.Int32` type in the runtime. You can use `System.Int32` in most of the places where you use `int`, but that isn't recommended because it makes the code tougher to read.

Table 3-2 describes the basic types. You can find all the runtime types in the `System` namespace of the .NET CLR.

Table 3-2. *Basic Data Types*

Type	Bytes	Runtime Type	Description
`byte`	1	`Byte`	Unsigned byte
`sbyte`	1	`SByte`	Signed byte
`short`	2	`Int16`	Signed short
`ushort`	2	`UInt16`	Unsigned short
`int`	4	`Int32`	Signed integer
`uint`	4	`UInt32`	Unsigned int
`long`	8	`Int64`	Signed big integer
`ulong`	8	`UInt64`	Unsigned big integer
`float`	4	`Single`	Floating-point number
`double`	8	`Double`	Double-precision floating-point number
`decimal`	8	`Decimal`	Fixed-precision number
`string`		`String`	Unicode string
`char`	2	`Char`	Unicode character
`bool`		`Boolean`	Boolean value

The distinction between basic (or built-in) types in C# is largely an artificial one, as user-defined types can operate in the same manner as the built-in ones. In fact, the only real difference between the built-in data types and user-defined data types is that it's possible to write literal values for the built-in types.

Data types are separated into value types and reference types. Value types are either stack allocated or allocated inline in a structure. Reference types are heap allocated.

Both reference and value types are derived from the ultimate base class `object`. In cases where a value type needs to act like an `object`, a wrapper that makes the value type look like a reference object is allocated on the heap, and the value type's value is copied into it. This process is known as *boxing*, and the reverse process is known as *unboxing*. Boxing and unboxing let you treat *any* type as an `object`. This allows the following to be written:

```
using System;
class Hello
{
    public static void Main(string[] args)
    {
        Console.WriteLine("Value is: {0}", 3);
    }
}
```

In this case, the integer `3` is boxed, and the `Int32.ToString()` function is called on the boxed value.

C# arrays can be declared in the multidimensional form or the jagged form. You can find more advanced data structures, such as stacks and hash tables, in the `System.Collections` namespace.

Classes, Structs, and Interfaces

In C#, the `class` keyword declares a reference (heap-allocated) type, and the `struct` keyword declares a value type. Structs are used for lightweight objects that need to act like the built-in types, and classes are used in all other cases. For example, the `int` type is a value type, and the `string` type is a reference type. Figure 3-1 details how these work.

```
int v = 123;
string s = "Hello There";
```

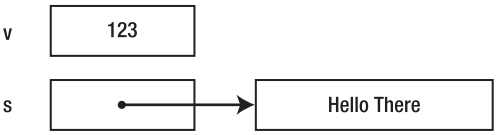

Figure 3-1. *Value and reference type allocation*

C# and the .NET runtime don't support multiple inheritance for classes but do support multiple implementation of interfaces.

Statements

The statements in C# are close to C++ statements, with a few modifications to make errors less likely and a few new statements. You can use the `foreach` statement to iterate over arrays and collections, the `lock` statement for mutual exclusion in threading scenarios, and the `checked` and `unchecked` statements for controlling overflow checking in arithmetic operations and conversions.

Enums

Enumerators declare a set of related constants—such as the colors that a control can take—in a clear and type-safe manner. For example:

```
enum Colors
{
    red,
    green,
    blue
}
```

Chapter 21 covers enumerators in more detail.

Delegates and Events

Delegates are a type-safe, object-oriented implementation of function pointers and are used in many situations where a component needs to call back to the component that's using it. They're used most heavily as the basis for events, which allow a delegate to easily be registered for an event. Chapter 23 discusses them.

The .NET Framework use delegates and events heavily.

Properties and Indexers

C# supports properties and indexers, which are useful for separating the interface of an object from the implementation of the object. Rather than allowing a user to access a field or array directly, a property or indexer allows a statement block to be specified to perform the access while still allowing the field or array usage. Here's a simple example:

```
using System;
class Circle
{
    public int Radius
    {
        get
        {
            return(radius);
        }
        set
        {
            radius = value;
            Draw();
        }
    }
    public void Draw()
    {
    }
    int radius;
}
class Test
{
    public static void Main()
    {
        Circle c = new Circle();
        c.Radius = 35;
    }
}
```

In this example, the `get` or `set` accessor is called when the property `Radius` is referenced.

Attributes

Attributes are used in C# and the .NET Framework to communicate declarative information from the writer of the code to other code that's interested in the information. You could use this to specify which fields of an object should be serialized, what transaction context to use when running an object, how to marshal fields to native functions, or how to display a class in a class browser.

Attributes are specified within square braces. A typical attribute usage might look like this:

```
[CodeReview("12/31/1999", Comment="Well done")]
```

Attribute information is retrieved at runtime through a process known as *reflection*. New attributes can be easily written, applied to elements of the code (such as classes, members, or parameters), and retrieved through reflection.

Developing in C#

To program in C#, you're going to need a way to build C# programs. You can do this with a command-line compiler, VS .NET, or a C# package for a programming editor.

The Command-Line Compiler

The simplest way to get started writing C# code is by using the .NET runtime and Framework's SDK. The SDK contains the .NET runtime and Framework and compilers for C#, Visual Basic .NET, the Managed Extensions to C++, and JScript .NET. You can download the framework from the following site:

```
http://msdn.microsoft.com/netframework/
```

Using the SDK is easy; write your code in an editor, compile it using `csc`, and then run it. But easy doesn't necessarily mean productive, however. When writing with just the SDK, you'll spend a lot of time looking at documentation and trying to figure out simple errors in your code.

You can find details about using the command-line compiler in Chapter 41.

Visual Studio .NET

Visual Studio .NET provides a full environment to make programming in C# easy and fun. Visual Studio versions are bound to framework versions, with Visual Studio .NET 2002 targeting .NET 1.0, Visual Studio .NET 2003 targeting .NET 1.1, and Visual Studio .NET 2005 targeting .NET 2.0.

Visual Studio .NET provides some real help for the developer.

The Editor

The most important feature of the Visual Studio .NET editor is the IntelliSense support. One of the challenges of learning C# and the .NET Framework is simply finding your way around the syntax of the language and object model of the .NET Framework. Autocompletion will help greatly in finding the proper method to use, and syntax checking will highlight the sections of code with errors.

The Form Designer

Both Windows Forms applications and Web Forms applications are written directly in code, without separate resource files, so it's possible to write either application without Visual Studio .NET.

Doing layout and setting properties by hand isn't very fun, however, so Visual Studio provides a Form Designer that makes it easy to add controls to a form, set the properties on the form, create event handlers, and so on.

The form editing is two ways; changes made in the Form Designer show up in the form code, and changes made in the form code show up in the designer.[2]

The Project System

The project system provides support for creating and building projects. There are predefined templates for most types of projects (Windows Forms, Web Forms, Console Application, Class Library, and so on).

The project system also provides support for deploying applications.

Class View

While the project system provides a view of the files in a project, Class View provides a view into the classes and other types in a project. Rather than writing the syntax for a property directly, for example, Class View provides a wizard to do it for you.

The Object Browser

The Object Browser provides the same sort of view that Class View does, but instead of looking at the code in the current project, it lets you browse the components in other assemblies that the project is using. If the documentation for a class is incomplete, the Object Browser can show the interface of the class as defined by the metadata for the component.

The Debugger

The debugger in Visual Studio .NET has been enhanced to provide cross-language debugging facilities. It's possible to debug from one language to the other, and it's also possible to debug code executing remotely on another machine.

Other Tools of Note

The SDK comes with a number of utility programs, which are detailed in the .NET Framework Tools section of the documentation.

ILDASM

IL Disassembler (ILDASM) is the most useful tool in the SDK. It can open an assembly, show all the types in the assembly, show what methods are defined for those types, and show the IL that was generated for that method.

---

2. Within reason.

This is useful in a number of ways. Like the Object Browser, it can be used to find out what's present in an assembly, but it can also be used to find out how a specific method is implemented. You can use this capability to answer some questions about C#.

If, for example, you want to know whether C# will concatenate constant strings at compile time, it's easy to test. First, create a short program:

```
using System;
class Test
{
    public static void Main()
    {
        Console.WriteLine("Hello " + "World");
    }
}
```

Second, after the program is compiled, use ILDASM to view the IL for `Main()`:

```
.method public hidebysig static void Main() cil managed
{
  .entrypoint
  // Code size       11 (0xb)
  .maxstack  8
  IL_0000:  ldstr      "Hello World"
  IL_0005:  call       void [mscorlib]System.Console::WriteLine(string)
  IL_000a:  ret
} // end of method Test::Main
```

Even without knowing the details of IL, it's pretty clear that the two strings are concatenated into a single string.

You can find the details of IL in `ILinstrset.doc`, which is in the SDK install folder, which is a subfolder of the main Visual Studio install folder.

You can use ILDASM on any assembly, which raises some questions over intellectual property. Although having code stored in IL makes disassemblers easier to write, it does create an issue that didn't exist before: x86 assembly language can also be disassembled and decoded. Various obfuscators exist to address this issue, and an entry-level obfuscator ships with some versions of Visual Studio. Check the Visual Studio Web site for the exact details.

NGEN

NGEN is a tool that compiles the MSIL code to native code when NGEN is executed. This is different from the standard .NET loading model where conversion to native code happens Just-in-Time (JIT) when the .NET assembly is loaded.

At first glance, this seems like a way to get around many of the disadvantages of the JIT approach; simply PreJIT the code, and then performance will be better and nobody will be able to decode the IL.

Unfortunately, things don't work that way. PreJIT is only a way to store the results of the compilation, but the metadata is still required to do class layout and support reflection. Further, the generated native code is valid only for a specific environment, and if configuration settings (such as the machine security policy) change, the runtime will switch back to the normal JIT.

Although PreJIT does eliminate the overhead of the JIT process, it also produces code that runs slightly slower because it requires a level of indirection that isn't required with the normal JIT.

So, the real benefit of PreJIT is to reduce the JIT overhead (and therefore the startup time) of a client application, and it isn't really terribly useful elsewhere.

.NET 2.0 fixes the problem of needing to load two physical DLLs for each PreJITed assembly and also introduces more optimizations during the PreJIT phase. For developers who tried and rejected NGEN in .NET 1.*x*, it's worth reevaluating for .NET 2.0.

CHAPTER 4

Exception Handling

In many programming books, exception handling warrants a chapter somewhat late in the book. In this book, however, it's near the front for a couple of reasons.

The first reason is that exception handling is deeply ingrained in the .NET runtime and is therefore common in C# code. C++ code can be written without using exception handling, but that's not an option in C#.

The second reason is that it allows the code examples to be better. If exception handling is late in the book, early code samples can't use it, and that means the examples can't be written using good programming practices.

Unfortunately, this means classes must be used without really introducing them. Read the following section for flavor; we'll cover the classes in detail in the next chapter.

What's Wrong with Return Codes?

Most programmers have probably written code that looks like this:

```
bool success = CallFunction();
if (!success)
{
    // process the error
}
```

This works okay, but every return value has to be checked for an error. If the previous code was written as

```
CallFunction();
```

any error return would be thrown away. That's where bugs come from.

Many different models exist for communicating status; some functions may return an `HRESULT`, some may return a Boolean value, and others may use some other mechanism.

In the .NET runtime world, exceptions are the fundamental method of handling error conditions. Exceptions are nicer than return codes because they can't be silently ignored.

Trying and Catching

To deal with exceptions, code needs to be organized a bit differently. The sections of code that might throw exceptions are placed in a `try` block, and the code to handle exceptions in the `try` block is placed in a `catch` block. Here's an example:

```
using System;
class Test
{
    static int Zero = 0;
    public static void Main()
    {
            // watch for exceptions here
        try
        {
            int j = 22 / Zero;
        }
            // exceptions that occur in try are transferred here
        catch (Exception e)
        {
            Console.WriteLine("Exception " + e.Message);
        }
        Console.WriteLine("After catch");
    }
}
```

The `try` block encloses an expression that will generate an exception. In this case, it will generate an exception known as `DivideByZeroException`. When the division takes place, the .NET runtime stops executing code and searches for a `try` block surrounding the code in which the exception took place. When it finds a `try` block, it then looks for associated `catch` blocks.

If it finds `catch` blocks, it picks the best one (more on how it determines which one is best in a minute) and executes the code within the `catch` block. The code in the `catch` block may process the event or rethrow it.

The example code catches the exception and writes out the message that's contained within the exception object.

The Exception Hierarchy

All C# exceptions derive from the class named `Exception`, which is part of the CLR.[1] When an exception occurs, the proper `catch` block is determined by matching the type of the exception to the name of the exception mentioned. A `catch` block with an exact match wins out over a more general exception. Let's return to the example:

1. This is true of .NET classes in general, but it might not hold true in some cases.

```
using System;
class Test
{
    static int Zero = 0;
    public static void Main()
    {
        try
        {
            int j = 22 / Zero;
        }
            // catch a specific exception
        catch (DivideByZeroException e)
        {
            Console.WriteLine("DivideByZero {0}", e);
        }
            // catch any remaining exceptions
        catch (Exception e)
        {
            Console.WriteLine("Exception {0}", e);
        }
    }
}
```

The `catch` block that catches the `DivideByZeroException` is the more specific match and is therefore the one that's executed. `catch` blocks always must be listed from most specific to least specific, so in this example, the two blocks couldn't be reversed.[2]

This example is a bit more complex:

```
using System;
class Test
{
    static int Zero = 0;
    static void AFunction()
    {
        int j = 22 / Zero;
            // the following line is never executed.
        Console.WriteLine("In AFunction()");
    }
    public static void Main()
    {
        try
        {
            AFunction();
        }
```

2. More specifically, a clause catching a derived exception can't be listed after a clause catching a base exception.

```
        catch (DivideByZeroException e)
        {
            Console.WriteLine("DivideByZero {0}", e);
        }
    }
}
```

What happens here?

When the division is executed, an exception is generated. The runtime starts searching for a `try` block in `AFunction()`, but it doesn't find one, so it jumps out of `AFunction()` and checks for a `try` in `Main()`. It finds one and then looks for a `catch` that matches. The `catch` block then executes.

Sometimes, there won't be any `catch` clauses that match.

```
using System;
class Test
{
    static int Zero = 0;
    static void AFunction()
    {
        try
        {
            int j = 22 / Zero;
        }
            // this exception doesn't match
        catch (ArgumentOutOfRangeException e)
        {
            Console.WriteLine("OutOfRangeException: {0}", e);
        }
        Console.WriteLine("In AFunction()");
    }
    public static void Main()
    {
        try
       {
            AFunction();
        }
            // this exception doesn't match
        catch (ArgumentException e)
        {
            Console.WriteLine("ArgumentException {0}", e);
        }
    }
}
```

Neither the `catch` block in `AFunction()` nor the `catch` block in `Main()` matches the exception that's thrown. When this happens, the exception is caught by the "last-chance" exception handler. The action taken by this handler depends on how the runtime is configured, but it will usually bring up a dialog box containing the exception information and halt the program.

Passing Exceptions on to the Caller

Sometimes you can't do much when an exception occurs; it really has to be handled by the calling function. You have three basic ways to deal with this, which are named based on their result in the caller: Caller Beware, Caller Confuse, and Caller Inform.

Caller Beware

The first way is to merely not catch the exception. This is sometimes the right design decision, but it could leave the object in an incorrect state, causing problems when the caller tries to use it later. It may also give insufficient information to the caller.

Caller Confuse

The second way is to catch the exception, do some cleanup, and then rethrow the exception:

```
using System;
public class Summer
{
    int    sum = 0;
    int    count = 0;
    float  average;
    public void DoAverage()
    {
        try
        {
            average = sum / count;
        }
        catch (DivideByZeroException e)
        {
            // do some cleanup here
            throw;
        }
    }
}
class Test
{
    public static void Main()
    {
        Summer summer = new Summer();
        try
        {
            summer.DoAverage();
        }
        catch (Exception e)
        {
            Console.WriteLine("Exception {0}", e);
        }
    }
}
```

This is usually the minimal bar for handling exceptions; an object should always maintain a valid state after an exception.

This is called *Caller Confuse* because although the object is in a valid state after the exception occurs, the caller often has little information to go on. In this case, the exception information says that a `DivideByZeroException` occurred somewhere in the called function, without giving any insight into the details of the exception or how it might be fixed.

Sometimes this is okay if the exception passes back obvious information.

Caller Inform

In *Caller Inform*, additional information is returned for the user. The caught exception is wrapped in an exception that has additional information. For example:

```
using System;
public class Summer
{
    int    sum = 0;
    int    count = 0;
    float  average;
    public void DoAverage()
    {
        try
        {
            average = sum / count;
        }
        catch (DivideByZeroException e)
        {
                // wrap exception in another one,
                // adding additional context.
                throw (new DivideByZeroException(
                    "Count is zero in DoAverage()", e));
        }
    }
}
public class Test
{
    public static void Main()
    {
        Summer summer = new Summer();
        try
        {
            summer.DoAverage();
        }
        catch (Exception e)
        {
            Console.WriteLine("Exception: {0}", e);
        }
    }
}
```

When the DivideByZeroException is caught in the DoAverage() function, it's wrapped in a new exception that gives the user additional information about what caused the exception. Usually the wrapper exception is the same type as the caught exception, but this might change depending on the model presented to the caller.

This program generates the following output:

```
Exception: System.DivideByZeroException: Count is zero in DoAverage()
  ---> System.DivideByZeroException
   at Summer.DoAverage()
   at Summer.DoAverage()
   at Test.Main()
```

Ideally, each function that wants to rethrow the exception will wrap it in an exception with additional contextual information.

User-Defined Exception Classes

One drawback of the last example is that the caller can't tell what exception happened in the call to DoAverage() by looking at the type of the exception. To know that the exception was because the count was zero, the expression message would have to be searched for the string Count is zero.

That would be pretty bad, since the user wouldn't be able to trust that the text would remain the same in later versions of the class, and the class writer wouldn't be able to change the text. In this case, a new exception class can be created:

```
using System;
public class CountIsZeroException: ApplicationException
{
    public CountIsZeroException()
    {
    }
    public CountIsZeroException(string message)
    : base(message)
    {
    }
    public CountIsZeroException(string message, Exception inner)
    : base(message, inner)
    {
    }
}
public class Summer
{
    int    sum = 0;
    int    count = 0;
    float  average;
    public void DoAverage()
```

```
    {
        if (count == 0)
            throw(new CountIsZeroException("Zero count in DoAverage"));
        else
            average = sum / count;
    }
}
class Test
{
    public static void Main()
    {
        Summer summer = new Summer();
        try
        {
            summer.DoAverage();
        }
        catch (CountIsZeroException e)
        {
            Console.WriteLine("CountIsZeroException: {0}", e);
        }
    }
}
```

`DoAverage()` now determines whether there would be an exception (whether `count` is zero) and, if so, creates a `CountIsZeroException` and throws it.

In this example, the `exception` class has three constructors, which is the recommended design pattern. It's important to follow this design pattern because if the constructor that takes the inner exception is missing, it won't be possible to wrap the exception with the same exception type; it could be wrapped only in something more general. If, in the previous example, the caller didn't have that constructor, a caught `CountIsZeroException` couldn't be wrapped in an exception of the same type, and the caller would have to choose between not catching the exception and wrapping it in a less-specific type.

Also notice that the `exception` class is derived from `ApplicationException`, which is the base of application-derived exceptions and therefore should be used for all exceptions defined in an application.

Finally

Sometimes, when writing a function, you'll need to do some cleanup before the function completes, such as closing a file. If an exception occurs, the cleanup could be skipped:

```
using System;
using System.IO;
class Processor
{
    int    count;
    int    sum;
```

```
    public int average;
    void CalculateAverage(int countAdd, int sumAdd)
    {
        count += countAdd;
        sum += sumAdd;
        average = sum / count;
    }
    public void ProcessFile()
    {
        FileStream f = new FileStream("data.txt", FileMode.Open);
        try
        {
            StreamReader t = new StreamReader(f);
            string    line;
            while ((line = t.ReadLine()) != null)
            {
                int count;
                int sum;
                count = Convert.ToInt32(line);
                line = t.ReadLine();
                sum = Convert.ToInt32(line);
                CalculateAverage(count, sum);
            }
                  }
            // always executed before function exit, even if an
            // exception was thrown in the try.
        finally
        {
            f.Close();
        }
    }
}
class Test
{
    public static void Main()
    {
        Processor processor = new Processor();
        try
        {
            processor.ProcessFile();
        }
        catch (Exception e)
        {
            Console.WriteLine("Exception: {0}", e);
        }
    }
}
```

This example walks through a file, reading a `count` and `sum` from a file and using it to accumulate an average. What happens, however, if the first `count` read from the file is a zero?

If this happens, the division in `CalculateAverage()` will throw a `DivideByZeroException`, which will interrupt the file-reading loop. If the programmer had written the function without thinking about exceptions, the call to `file.Close()` would have been skipped, and the file would have remained open.

The code inside the `finally` block is guaranteed to execute before the exit of the function, whether or not there is an exception. By placing the `file.Close()` call in the `finally` block, the file will always be closed.

Efficiency and Overhead

In languages without garbage collection, adding exception handling is expensive, since all objects within a function must be tracked to make sure they're properly destroyed at any time that an exception could be thrown. The required tracking code adds both execution time and code size to a function.

In C#, however, objects are tracked by the garbage collector rather than the compiler, so exception handling is inexpensive to implement and imposes little runtime overhead on the program when the exceptional case doesn't occur.

Design Guidelines

You should use exceptions to communicate exceptional conditions. Don't use them to communicate events that are expected, such as reaching the end of a file. In the normal operation of a class, no exceptions should be thrown.

Conversely, don't use return values to communicate information that would be better contained in an exception.

If a good predefined exception in the `System` namespace describes the exception condition—one that will make sense to the users of the class—use that one rather than defining a new exception class, and put specific information in the message. If the user might want to differentiate one case from others where that same exception might occur, then that would be a good place for a new exception class.

Finally, if code catches an exception that it isn't going to handle, consider whether it should wrap that exception with additional information before rethrowing it.

CHAPTER 5

Classes 101

Classes are the heart of any application in an object-oriented language. This chapter is broken into several sections. The first section describes the parts of C# you'll use often, and the later sections describe features you won't use as often, depending on what kind of code you're writing.

A Simple Class

A C# class can be very simple:

```
class VerySimple
{
    int    simpleValue = 0;
}
class Test
{
    public static void Main()
    {
        VerySimple vs = new VerySimple();
    }
}
```

This class is a container for a single integer. Because the integer is declared without specifying how accessible it is, it's private to the `VerySimple` class and can't be referenced outside the class. The `private` modifier could be specified to state this explicitly.

The integer `simpleValue` is a member of the class; there can be many different types of members.

In the `Main()` function, the system creates the instance in heap memory and returns a reference to the instance. A reference is simply a way to refer to an instance.[1]

There's no need to specify when an instance is no longer needed. In the preceding example, as soon as the `Main()` function completes, the reference to the instance will no longer exist. If the reference hasn't been stored elsewhere, the instance will then be available for reclamation by the garbage collector. The garbage collector will reclaim the memory that was allocated when necessary.[2]

1. For those of you used to pointers, a reference is a pointer that you can only assign to and dereference.
2. Chapter 38 discusses the garbage collector used in the .NET runtime.

This is all very nice, but this class doesn't do anything useful because the integer isn't accessible. Here's a more useful example:[3]

```
using System;
class Point
{
        // constructor
    public Point(int x, int y)
    {
        this.x = x;
        this.y = y;
    }

        // member fields
    public int x;
    public int y;
}

class Test
{
    public static void Main()
    {
        Point myPoint = new Point(10, 15);
        Console.WriteLine("myPoint.x {0}", myPoint.x);
        Console.WriteLine("myPoint.y {0}", myPoint.y);
    }
}
```

This example has a class named `Point`, with two integers in the class named `x` and `y`. These members are public, which means their values can be accessed by any code that uses the class.

In addition to the data members, the class has a constructor, which is a special function that's called to help construct an instance of the class. The constructor takes two integer parameters.

In this constructor, a special variable called `this` is used; the `this` variable is available within all member functions and always refers to the current instance.

In member functions, the compiler searches local variables and parameters for a name before searching instance members. When referring to an instance variable with the same name as a parameter, you must use the `this.<name>` syntax.

In this constructor, `x` by itself refers to the parameter named `x`, and `this.x` refers to the integer member named `x`.

In addition to the `Point` class, you'll see a `Test` class that contains a `Main` function that's called to start the program. The `Main` function creates an instance of the `Point` class, which will allocate memory for the object and then call the constructor for the class. The constructor will set the values for `x` and `y`.

The remainder of the lines of `Main()` print the values of `x` and `y`.

---

3. If you were really going to implement your own point class, you'd probably want it to be a value type (struct) rather than a reference type (class).

In this example, the data fields are accessed directly. This is usually a bad idea; it means that users of the class depend upon the names of fields, which constrains the modifications that can be made later.

In C#, rather than writing a member function to access a private value, you'd use a property, which gives the benefits of a member function while retaining the user model of a field. See Chapter 19 for more information.

Member Functions

The constructor in the previous example is an example of a member function; it's a piece of code that's called on an instance of the object. Constructors can be called automatically only when an instance of an object is created with `new`.

You can declare other member functions as follows:

```
using System;
class Point
{
    public Point(int x, int y)
    {
        this.x = x;
        this.y = y;
    }

        // accessor functions
    public int GetX() {return(x);}
    public int GetY() {return(y);}

        // variables now private
    int x;
    int y;
}

class Test
{
    public static void Main()
    {
        Point myPoint = new Point(10, 15);
        Console.WriteLine("myPoint.X {0}", myPoint.GetX());
        Console.WriteLine("myPoint.Y {0}", myPoint.GetY());
    }
}
```

ref and out Parameters

Having to call two member functions to get the values may not always be convenient, so it'd be nice to be able to get both values with a single function call. There's only one return value, however.

One solution is to use reference (or ref) parameters so you can modify the values of the parameters passed into the member function:

```
// error
using System;
class Point
{
    public Point(int x, int y)
    {
        this.x = x;
        this.y = y;
    }
        // get both values in one function call
    public void GetPoint(ref int x, ref int y)
    {
        x = this.x;
        y = this.y;
    }

    int x;
    int y;
}

class Test
{
    public static void Main()
    {
        Point myPoint = new Point(10, 15);
        int    x;
        int     y;

            // illegal
        myPoint.GetPoint(ref x, ref y);
        Console.WriteLine("myPoint({0}, {1})", x, y);
    }
}
```

In this code, the parameters have been declared using the ref keyword, as has the call to the function.

This code should work; however, when compiled, it generates an error message that says that uninitialized values were used for the ref parameters x and y. This means variables were passed into the function before having their values set, and the compiler won't allow the values of uninitialized variables to be exposed.

You have two ways to get around this. The first is to initialize the variables when they're declared:

```
using System;
class Point
{
    public Point(int x, int y)
    {
        this.x = x;
        this.y = y;
    }

    public void GetPoint(ref int x, ref int y)
    {
        x = this.x;
        y = this.y;
    }

    int x;
    int y;
}

class Test
{
    public static void Main()
    {
        Point myPoint = new Point(10, 15);
        int x = 0;
        int y = 0;

        myPoint.GetPoint(ref x, ref y);
        Console.WriteLine("myPoint({0}, {1})", x, y);
    }
}
```

The code now compiles, but the variables are initialized to zero only to be overwritten in the call to `GetPoint()`. For C#, another option is to change the definition of the function `GetPoint()` to use an `out` parameter rather than a `ref` parameter:

```
using System;
class Point
{
    public Point(int x, int y)
    {
        this.x = x;
        this.y = y;
    }
```

```
    public void GetPoint(out int x, out int y)
    {
        x = this.x;
        y = this.y;
    }

    int x;
    int y;
}

class Test
{
    public static void Main()
    {
        Point myPoint = new Point(10, 15);
        int    x;
        int     y;

        myPoint.GetPoint(out x, out y);
        Console.WriteLine("myPoint({0}, {1})", x, y);
    }
}
```

`Out` parameters are like `ref` parameters except that an uninitialized variable can be passed to them and the call is made with `out` rather than `ref`.[4]

Overloading

Sometimes it may be useful to have two functions that do the same thing but take different parameters. This is especially common for constructors, when there may be several ways to create a new instance. For example:

```
class Point
{
        // create a new point from x and y values
    public Point(int x, int y)
    {
        this.x = x;
        this.y = y;
    }
```

4. From the perspective of other .NET languages, there's no difference between `ref` and `out` parameters. A C# program calling this function will see the parameters as `out` parameters, but other languages will see them as `ref` parameters.

```
        // create a point from an existing point
    public Point(Point p)
    {
        this.x = p.x;
        this.y = p.y;
    }

    int x;
    int y;
}

class Test
{
    public static void Main()
    {
        Point myPoint = new Point(10, 15);
        Point mySecondPoint = new Point(myPoint);
    }
}
```

The class has two constructors: one that can be called with x and y values and one that can be called with another point. The `Main()` function uses both constructors: one to create an instance from an x and y value and another to create an instance from an already-existing instance.[5]

When an overloaded function is called, the compiler chooses the proper function by matching the parameters in the call to the parameters declared for the function.

5. This function may look like a C++ copy constructor, but the C# language doesn't use such a concept. A constructor such as this must be called explicitly.

CHAPTER 6

Base Classes and Inheritance

As discussed in Chapter 1, it makes sense to derive one class from another if the derived class is an example of the base class.

The Engineer Class

The following class implements `Engineer` and methods to handle billing for `Engineer`:

```
using System;
class Engineer
{
        // constructor
    public Engineer(string name, float billingRate)
    {
        this.name = name;
        this.billingRate = billingRate;
    }
        // figure out the charge based on engineer's rate
    public float CalculateCharge(float hours)
    {
        return(hours * billingRate);
    }
        // return the name of this type
    public string TypeName()
    {
        return("Engineer");
    }

    private string name;
    protected float billingRate;
}
```

```
class Test
{
    public static void Main()
    {
        Engineer engineer = new Engineer("Hank", 21.20F);
        Console.WriteLine("Name is: {0}", engineer.TypeName());
    }
}
```

`Engineer` will serve as a base class for this scenario. It contains the private field `name` and the protected field `billingRate`. The `protected` modifier grants the same access as `private`; however, classes that are derived from this class also have access to the field. `Protected` is therefore used to give classes that derive from this class access to a field.

`Protected` access allows other classes to depend upon the internal implementation of the class and therefore should be granted only when necessary. In the example, the `billingRate` member can't be renamed, since derived classes may access it. It's often a better design choice to use a protected property.

The `Engineer` class also has a member function that can be used to calculate the charge based on the number of hours of work done.

Simple Inheritance

A `CivilEngineer` is a type of engineer and therefore can be derived from the `Engineer` class:

```
using System;
class Engineer
{
    public Engineer(string name, float billingRate)
    {
        this.name = name;
        this.billingRate = billingRate;
    }

    public float CalculateCharge(float hours)
    {
        return(hours * billingRate);
    }

    public string TypeName()
    {
        return("Engineer");
    }

    private string name;
    protected float billingRate;
}
```

```
class CivilEngineer: Engineer
{
    public CivilEngineer(string name, float billingRate) :
        base(name, billingRate)
    {
    }
        // new function, because it's different than the
        // base version
    public new float CalculateCharge(float hours)
    {
        if (hours < 1.0F)
            hours = 1.0F;        // minimum charge.
        return(hours * billingRate);
    }
        // new function, because it's different than the
        // base version
    public new string TypeName()
    {
        return("Civil Engineer");
    }
}
class Test
{
    public static void Main()
    {
        Engineer    e = new Engineer("George", 15.50F);
        CivilEngineer    c = new CivilEngineer("Sir John", 40F);

        Console.WriteLine("{0} charge = {1}",
                    e.TypeName(),
                    e.CalculateCharge(2F));
        Console.WriteLine("{0} charge = {1}",
                    c.TypeName(),
                    c.CalculateCharge(0.75F));
    }
}
```

Because the `CivilEngineer` class derives from `Engineer`, it inherits all the data members of the class (though the `name` member can't be accessed because it's private), and it also inherits the `CalculateCharge()` member function.

Constructors can't be inherited, so a separate one is written for `CivilEngineer`. The constructor doesn't have anything special to do, so it calls the constructor for `Engineer`, using the `base` syntax. If you omitted the call to the `base` class constructor, the compiler would call the base class constructor with no parameters.

`CivilEngineer` has a different way to calculate charges; the minimum charge is for one hour of time, so there's a new version of `CalculateCharge()`.

The example, when run, yields the following output:

```
Engineer Charge = 31
Civil Engineer Charge = 40
```

Arrays of Engineers

This works fine in the early years, when there are only a few employees. As the company grows, it's easier to deal with an array of engineers.

Because `CivilEngineer` is derived from `Engineer`, an array of type `Engineer` can hold either type. This example has a different `Main()` function, putting the engineers into an array:

```
using System;
class Engineer
{
    public Engineer(string name, float billingRate)
    {
        this.name = name;
        this.billingRate = billingRate;
    }

    public float CalculateCharge(float hours)
    {
        return(hours * billingRate);
    }

    public string TypeName()
    {
        return("Engineer");
    }

    private string name;
    protected float billingRate;
}
class CivilEngineer: Engineer
{
    public CivilEngineer(string name, float billingRate) :
        base(name, billingRate)
    {
    }

    public new float CalculateCharge(float hours)
    {
        if (hours < 1.0F)
            hours = 1.0F;        // minimum charge.
        return(hours * billingRate);
    }
```

```
    public new string TypeName()
    {
        return("Civil Engineer");
    }
}
class Test
{
    public static void Main()
    {
            // create an array of engineers
        Engineer[] earray = new Engineer[2];
        earray[0] = new Engineer("George", 15.50F);
        earray[1] = new CivilEngineer("Sir John", 40F);

        Console.WriteLine("{0} charge = {1}",
                    earray[0].TypeName(),
                    earray[0].CalculateCharge(2F));
        Console.WriteLine("{0} charge = {1}",
                    earray[1].TypeName(),
                    earray[1].CalculateCharge(0.75F));
    }
}
```

This version yields the following output:

```
Engineer Charge = 31
Engineer Charge = 30
```

That's not right.

Because `CivilEngineer` is derived from `Engineer`, an instance of `CivilEngineer` can be used wherever an instance of `Engineer` is required.

When the engineers were placed into the array, the fact that the second engineer was really a `CivilEngineer` rather than an `Engineer` was lost. Because the array is an array of `Engineer`, when `CalculateCharge()` is called, the version from `Engineer` is called.

What's needed is a way to correctly identify the type of an engineer. You can do this by having a field in the `Engineer` class that denotes what type it is. Rewriting the classes with an `enum` field to denote the type of the engineer gives the following example:

```
using System;
enum EngineerTypeEnum
{
    Engineer,
    CivilEngineer
}
class Engineer
{
    public Engineer(string name, float billingRate)
```

```
    {
        this.name = name;
        this.billingRate = billingRate;
        type = EngineerTypeEnum.Engineer;
    }

    public float CalculateCharge(float hours)
    {
        if (type == EngineerTypeEnum.CivilEngineer)
        {
            CivilEngineer c = (CivilEngineer) this;
            return(c.CalculateCharge(hours));
        }
        else if (type == EngineerTypeEnum.Engineer)
            return(hours * billingRate);
        return(0F);
    }

    public string TypeName()
    {
        if (type == EngineerTypeEnum.CivilEngineer)
        {
            CivilEngineer c = (CivilEngineer) this;
            return(c.TypeName());
        }
        else if (type == EngineerTypeEnum.Engineer)
            return("Engineer");
        return("No Type Matched");
    }

    private string name;
    protected float billingRate;
    protected EngineerTypeEnum type;
}

class CivilEngineer: Engineer
{
    public CivilEngineer(string name, float billingRate) :
        base(name, billingRate)
    {
        type = EngineerTypeEnum.CivilEngineer;
    }
```

```
    public new float CalculateCharge(float hours)
    {
        if (hours < 1.0F)
            hours = 1.0F;        // minimum charge.
        return(hours * billingRate);
    }

    public new string TypeName()
    {
        return("Civil Engineer");
    }
}
class Test
{
    public static void Main()
    {
        Engineer[] earray = new Engineer[2];
        earray[0] = new Engineer("George", 15.50F);
        earray[1] = new CivilEngineer("Sir John", 40F);

        Console.WriteLine("{0} charge = {1}",
                    earray[0].TypeName(),
                    earray[0].CalculateCharge(2F));
        Console.WriteLine("{0} charge = {1}",
                    earray[1].TypeName(),
                    earray[1].CalculateCharge(0.75F));
    }
}
```

By looking at the `type` field, the functions in `Engineer` can determine the real type of the object and call the appropriate function.

The output of the code is as expected:

```
Engineer Charge = 31
Civil Engineer Charge = 40
```

Unfortunately, the base class has now become much more complicated; for every function that cares about the type of a class, there's code to check all the possible types and call the correct function. That's a lot of extra code, and it'd be untenable if there were 50 kinds of engineers.

Worse is that the base class needs to know the names of all the derived classes for it to work. If the owner of the code needs to add support for a new engineer, the base class must be modified. If a user who doesn't have access to the base class needs to add a new type of engineer, it won't work at all.

Virtual Functions

To make this work cleanly, object-oriented languages allow a function to be specified as virtual. *Virtual* means that when a call to a member function is made, the compiler should look at the real type of the object (not just the type of the reference) and call the appropriate function based on that type.

With that in mind, you can modify the example as follows:

```
using System;
class Engineer
{
    public Engineer(string name, float billingRate)
    {
        this.name = name;
        this.billingRate = billingRate;
    }
        // function now virtual
    virtual public float CalculateCharge(float hours)
    {
        return(hours * billingRate);
    }
        // function now virtual
    virtual public string TypeName()
    {
        return("Engineer");
    }

    private string name;
    protected float billingRate;
}

class CivilEngineer: Engineer
{
    public CivilEngineer(string name, float billingRate) :
        base(name, billingRate)
    {
    }
        // overrides function in Engineer
    override public float CalculateCharge(float hours)
    {
        if (hours < 1.0F)
            hours = 1.0F;        // minimum charge.
        return(hours * billingRate);
    }
```

```
        // overrides function in Engineer
    override public string TypeName()
    {
        return("Civil Engineer");
    }
}
class Test
{
    public static void Main()
    {
        Engineer[] earray = new Engineer[2];
        earray[0] = new Engineer("George", 15.50F);
        earray[1] = new CivilEngineer("Sir John", 40F);

        Console.WriteLine("{0} charge = {1}",
                    earray[0].TypeName(),
                    earray[0].CalculateCharge(2F));
        Console.WriteLine("{0} charge = {1}",
                    earray[1].TypeName(),
                    earray[1].CalculateCharge(0.75F));
    }
}
```

The `CalculateCharge()` and `TypeName()` functions are now declared with the virtual keyword in the base class, and that's all the base class has to know. It needs no knowledge of the derived types, other than to know that each derived class can override `CalculateCharge()` and `TypeName()`, if desired. In the derived class, the functions are declared with the `override` keyword, which means they're the same function that was declared in the base class. If the `override` keyword is missing, the compiler will assume that the function is unrelated to the base class's function, and virtual dispatching won't function.[1]

Running this example leads to the expected output:

```
Engineer Charge = 31
Civil Engineer Charge = 40
```

When the compiler encounters a call to `TypeName()` or `CalculateCharge()`, it goes to the definition of the function and notes that it's a virtual function. Instead of generating code to call the function directly, it writes a bit of dispatch code that at runtime will look at the real type of the object and call the function associated with the real type, rather than just the type of the reference. This allows the correct function to be called even if the class wasn't implemented when the caller was compiled.

For example, if some payroll processing code stored an array of `Engineer`, a new class derived from `Engineer` could be added to the system without having to modify or recompile the payroll code.

1. For a discussion of why this works this way, see Chapter 11.

The virtual dispatch has a small amount of overhead, so you shouldn't use it unless needed. A JIT could, however, notice that there were no derived classes from the class on which the function call was made and convert the virtual dispatch to a straight call.

Abstract Classes

The approach used so far has one small problem. A new class doesn't have to implement the `TypeName()` function, since it can inherit the implementation from `Engineer`. This makes it easy for a new class of engineer to have the wrong name associated with it.

If the `ChemicalEngineer` class is added, the example looks like this:

```
using System;
class Engineer
{
    public Engineer(string name, float billingRate)
    {
        this.name = name;
        this.billingRate = billingRate;
    }

    virtual public float CalculateCharge(float hours)
    {
        return(hours * billingRate);
    }

    virtual public string TypeName()
    {
        return("Engineer");
    }

    private string name;
    protected float billingRate;
}
class ChemicalEngineer: Engineer
{
    public ChemicalEngineer(string name, float billingRate) :
        base(name, billingRate)
    {
    }

    // overrides mistakenly omitted
}
```

```
class Test
{
    public static void Main()
    {
        Engineer[] earray = new Engineer[2];
        earray[0] = new Engineer("George", 15.50F);
        earray[1] = new ChemicalEngineer("Dr. Curie", 45.50F);

        Console.WriteLine("{0} charge = {1}",
                    earray[0].TypeName(),
                    earray[0].CalculateCharge(2F));
        Console.WriteLine("{0} charge = {1}",
                    earray[1].TypeName(),
                    earray[1].CalculateCharge(0.75F));
    }
}
```

The `ChemicalEngineer` class will inherit the `CalculateCharge()` function from `Engineer`, which might be correct, but it will also inherit `TypeName()`, which is definitely wrong. What's needed is a way to force `ChemicalEngineer` to implement `TypeName()`.

You can do this by changing `Engineer` from a normal class to an abstract class. In this abstract class, the `TypeName()` member function is marked as an abstract function, which means all classes that derive from `Engineer` will be required to implement the `TypeName()` function.

An abstract class defines a contract that derived classes are expected to follow.[2] Because an abstract class is missing "required" functionality, it can't be instantiated, which for the example means that instances of the `Engineer` class can't be created. So that there are still two distinct types of engineers, the `ChemicalEngineer` class has been added.

Abstract classes behave like normal classes except for one or more member functions that are marked as abstract. For example:

```
using System;
abstract class Engineer
{
    public Engineer(string name, float billingRate)
    {
        this.name = name;
        this.billingRate = billingRate;
    }

    virtual public float CalculateCharge(float hours)
    {
        return(hours * billingRate);
    }
```

2. You can achieve a similar effect by using interfaces. See Chapter 10 for a comparison of the two techniques.

```
    abstract public string TypeName();

    private string name;
    protected float billingRate;
}

class CivilEngineer: Engineer
{
    public CivilEngineer(string name, float billingRate) :
        base(name, billingRate)
    {
    }

    override public float CalculateCharge(float hours)
    {
        if (hours < 1.0F)
            hours = 1.0F;        // minimum charge.
        return(hours * billingRate);
    }
        // This override is required, or an error is generated.
    override public string TypeName()
    {
        return("Civil Engineer");
    }
}

class ChemicalEngineer: Engineer
{
    public ChemicalEngineer(string name, float billingRate) :
        base(name, billingRate)
    {
    }

    override public string TypeName()
    {
        return("Chemical Engineer");
    }
}
class Test
{
    public static void Main()
    {
        Engineer[] earray = new Engineer[2];
        earray[0] = new CivilEngineer("Sir John", 40.0F);
        earray[1] = new ChemicalEngineer("Dr. Curie", 45.0F);
```

```
        Console.WriteLine("{0} charge = {1}",
                    earray[0].TypeName(),
                    earray[0].CalculateCharge(2F));
        Console.WriteLine("{0} charge = {1}",
                    earray[1].TypeName(),
                    earray[1].CalculateCharge(0.75F));
    }
}
```

The `Engineer` class has been changed by the addition of `abstract` before the class, which indicates the class is abstract (in other words, has one or more abstract functions) and the addition of `abstract` before the `TypeName()` virtual function. The use of `abstract` on the virtual function is the important one; the one before the name of the class makes it clear that the class is abstract, since the abstract function could easily be buried amongst the other functions.

The implementation of `CivilEngineer` is identical, except that now the compiler will check to make sure that `TypeName()` is implemented by both `CivilEngineer` and `ChemicalEngineer`.

Sealed Classes and Methods

Sealed classes prevent a class from being used as a base class. They're primarily useful to prevent unintended derivation. For example:

```
// error
sealed class MyClass
{
    MyClass() {}
}
class MyNewClass : MyClass
{
}
```

This fails because `MyNewClass` can't use `MyClass` as a base class because `MyClass` is sealed.

Sealed classes are useful in cases where a class isn't designed with derivation in mind or where derivation could cause the class to break. The `System.String` class is sealed because strict requirements define how the internal structure must operate, and a derived class could easily break those rules.

A sealed method lets a class override a virtual function and prevents a derived class from overriding that same function. In other words, having sealed on a virtual method stops virtual dispatching. This is rarely useful, so sealed methods are rare.

CHAPTER 7

Member Accessibility and Overloading

One of the important decisions to make when designing an object is how accessible to make the members. In C#, you can control accessibility in several ways.

Class Accessibility

The coarsest level at which accessibility can be controlled is at the class. In most cases, the only valid modifiers on a class are `public`, which means everybody can see the class, and `internal`. The exception to this is nesting classes inside of other classes, which is a bit more complicated and is covered in Chapter 8.

The `internal` modifier is a way of granting access to a wider set of classes without granting access to everybody, and it's most often used when writing helper classes that should be hidden from the ultimate user of the class. In the .NET runtime world, `internal` equates to allowing access to all classes that are in the same assembly as this class.

Note In the C++ world, such accessibility is usually granted by using *friends*, which provide access to a specific class. The `friend` specifier provides greater granularity in specifying who can access a class, but in practice the access provided by `internal` is sufficient. In general, all classes should be `internal` unless users need to be able to access them.

Using internal on Members

You can also use the `internal` modifier on a member, which then allows that member to be accessible from classes in the same assembly as itself but not from classes outside the assembly.

This is especially useful when several public classes need to cooperate but some of the shared members shouldn't be exposed to the general public. Consider the following example:

```
public class DrawingObjectGroup
{
    public DrawingObjectGroup()
    {
        objects = new DrawingObject[10];
        objectCount = 0;
    }
    public void AddObject(DrawingObject obj)
    {
        if (objectCount < 10)
        {
            objects[objectCount] = obj;
            objectCount++;
        }
    }
    public void Render()
    {
        for (int i = 0; i < objectCount; i++)
        {
            objects[i].Render();
        }
    }

    DrawingObject[]    objects;
    int            objectCount;
}
public class DrawingObject
{
    internal void Render() {}
}
class Test
{
    public static void Main()
    {
        DrawingObjectGroup group = new DrawingObjectGroup();
        group.AddObject(new DrawingObject());
    }
}
```

Here, the `DrawingObjectGroup` object holds up to ten drawing objects. It's valid for the user to have a reference to a `DrawingObject`, but it'd be invalid for the user to call `Render()` for that object, so this is prevented by making the `Render()` function internal.

■ **Tip** This code wouldn't make sense in a real program. The .NET CLR has a number of collection classes that would make this sort of code much more straightforward and less error-prone.

---

internal protected

To provide some extra flexibility in how a class is defined, you can use the `internal protected` modifier to indicate that a member can be accessed from either a class that could access it through the `internal` access path or a class that could access it through a `protected` access path. In other words, `internal protected` allows `internal` or `protected` access.

Note that there's no way to specify `internal` and `protected` in C#, though an internal class with a protected member will provide that level of access.

The Interaction of Class and Member Accessibility

Class and member accessibility modifiers must both be satisfied for a member to be accessible. The accessibility of members is limited by the class so that it doesn't exceed the accessibility of the class.

Consider the following situation:

```
internal class MyHelperClass
{
    public void PublicFunction() {}
    internal void InternalFunction() {}
    protected void ProtectedFunction() {}
}
```

If you declared this class as a public class, the accessibility of the members would be the same as the stated accessibility; in other words, `PublicFunction()` would be public, `InternalFunction()` would be internal, and `ProtectedFunction()` would be protected.

Because the class is internal, however, the `public` on `PublicFunction()` is reduced to `internal`.

Method Overloading

When a single-named function has several overloaded methods, the C# compiler uses method overloading rules to determine which function to call.

In general, the rules are fairly straightforward, but the details can be somewhat complicated. Here's a simple example:

```
Console.WriteLine("Ummagumma");
```

To resolve this, the compiler will look at the `Console` class and find all methods that take a single parameter. It will then compare the type of the argument (`string` in this case) with the type of the parameter for each method, and if it finds a single match, that's the function to call. If it finds no matches, a compile-time error is generated. If it finds more than one match, things are a bit more complicated (see the "Better Conversions" section).

For an argument to match a parameter, it must fit one of the following cases:

- The argument type and the parameter type are the same type.
- An implicit conversion exists from the argument type to the parameter type, *and* the argument isn't passed using `ref` or `out`.

Note that in the previous description, the return type of a function isn't mentioned. That's because for C#—and for the .NET CLR—overloading based on return type isn't allowed.[1] Additionally, because `out` is a C#-only construct (it looks like `ref` to other languages), there can't be a `ref` overload and an `out` overload that differ only in their `ref` and `out`ness. There can, however, be a `ref` or `out` overload and a pass-by-value overload.

Method Hiding

When determining the set of methods to consider, the compiler will walk up the inheritance tree until it finds a method that's applicable and then perform overload resolution at that level in the inheritance hierarchy only; it won't consider functions declared at different levels of the hierarchy. Consider the following example:

```
using System;
public class Base
{
    public void Process(short value)
    {
        Console.WriteLine("Base.Process(short): {0}", value);
    }
}
public class Derived: Base
{
    public void Process(int value)
    {
        Console.WriteLine("Derived.Process(int): {0}", value);
    }

    public void Process(string value)
    {
        Console.WriteLine("Derived.Process(string): {0}", value);
    }
}
class Test
{
    public static void Main()
    {
        Derived d = new Derived();
        short i = 12;
        d.Process(i);
        ((Base) d).Process(i);
    }
}
```

1. In other words, C++ covariant return types aren't supported.

This example generates the following output:

```
Derived.Process(int): 12
Base.Process(short): 12
```

A quick look at this code might lead one to suspect that the `d.Process(i)` call would call the base class function because that version takes a `short`, which matches exactly. But according to the rules, once the compiler has determined that `Derived.Process(int)` is a match, it doesn't look any further up the hierarchy; therefore, `Derived.Process(int)` is the function called.

To call the base class function requires an explicit cast to the base class because the derived function hides the base class version.

Better Conversions

In some situations, there are multiple matches based on the simple rule mentioned previously. When this happens, a few rules determine which situation is considered better, and if there is a single one that's better than all the others, it's the one called.[2]

The three rules are as follows:

- An exact match of type is preferred over one that requires a conversion.
- If an implicit conversion exists from one type to another, and there's no implicit conversion the other direction, the type that has the implicit conversion is preferred.
- If the argument is a signed integer type, a conversion to another signed integer type is preferred over one to an unsigned integer type.

The first and third rules don't require a lot of explanation. The second, however, seems a bit more complex. An example should make it clearer:

```
using System;
public class MyClass
{
    public void Process(long value)
    {
        Console.WriteLine("Process(long): {0}", value);
    }
    public void Process(short value)
    {
        Console.WriteLine("Process(short): {0}", value);
    }
}
```

2. The rules for this are detailed in the C# Language Reference; consult the MSDN documentation.

```
class Test
{
    public static void Main()
    {
        MyClass myClass = new MyClass();

        int i = 12;
        myClass.Process(i);

        sbyte s = 12;
        myClass.Process(s);
    }
}
```

This example generates the following output:

```
Process(long): 12
Process(short): 12
```

In the first call to `Process()`, an `int` is passed as an argument. This matches the `long` version of the function because there's an implicit conversion from `int` to `long` and no implicit conversion from `int` to `short`.

In the second call, however, there are implicit conversions from `sbyte` to `short` or `long`. In this case, the second rule applies. There's an implicit conversion from `short` to `long`, and there isn't one from `long` to `short`; therefore, the version that takes a `short` is preferred.

Variable-Length Parameter Lists

It's sometimes useful to define a parameter to take a variable number of parameters. (`Console.WriteLine()` is a good example.) C# allows such support to be easily added:

```
using System;
class Port
{
        // version with a single object parameter
    public void Write(string label, object arg)
    {
        WriteString(label);
        WriteString(arg.ToString());
    }
```

```
        // version with an array of object parameters
    public void Write(string label, params object[] args)
    {
        WriteString(label);
        foreach (object o in args)
        {
            WriteString(o.ToString());
        }
    }
    void WriteString(string str)
    {
            // writes string to the port here
        Console.WriteLine("Port debug: {0}", str);
    }
}

class Test
{
    public static void Main()
    {
        Port    port = new Port();
        port.Write("Single Test", "Port ok");
        port.Write("Port Test: ", "a", "b", 12, 14.2);
        object[] arr = new object[4];
        arr[0] = "The";
        arr[1] = "answer";
        arr[2] = "is";
        arr[3] = 42;
        port.Write("What is the answer?", arr);
    }
}
```

The `params` keyword on the last parameter changes the way the compiler looks up functions. When it encounters a call to that function, it first checks to see if there's an exact match for the function. The first function call matches:

```
public void Write(string, object arg)
```

Similarly, the third function passes an object array, and it matches:

```
public void Write(string label, params object[] args)
```

Things get interesting for the second call. The definition with the object parameter doesn't match, but neither does the one with the object array.

When both of these matches fail, the compiler notices that the `params` keyword is present, and it then tries to match the parameter list by removing the array part of the `params` parameter and duplicating that parameter until there are the same number of parameters.

If this results in a function that matches, it then writes the code to create the object array. In other words, the following line:

```
port.Write("Port Test: ", "a", "b", 12, 14.2);
```

is rewritten as follows:

```
object[] temp = new object[4];
temp[0] = "a";
temp[1] = "b";
temp[2] = 12;
temp[3] = 14.2;
port.Write("Port Test: ", temp);
```

In this example, the `params` parameter was an `object` array, but it can be an array of any type.

In addition to the version that takes the array, it usually makes sense to provide one or more specific versions of the function. This is useful both for efficiency (so the object array doesn't have to be created) and so languages that don't support the `params` syntax don't have to use the object array for all calls. Overloading a function with versions that take one, two, and three parameters, plus a version that takes an array, is a good rule of thumb.

CHAPTER 8

Other Class Details

This chapter discusses some of the miscellaneous issues of classes, including constructors, nesting, and overloading rules.

Nested Classes

Sometimes it's convenient to nest classes within other classes, such as when a helper class is used by only one other class. The accessibility of the nested class follows similar rules to the ones outlined in Chapter 7 for the interaction of class and member modifiers. As with members, the accessibility modifier on a nested class defines what accessibility the nested class has outside the nested class. Just as a private field is visible only within a class, a private nested class is also visible only from within the class that contains it.

In the following example, the `Parser` class has a `Token` class that it uses internally. Without using a nested class, it might be written as follows:

```
public class Parser
{
    Token[]    tokens;
}
public class Token
{
    string name;
}
```

In this example, both the `Parser` and `Token` classes are publicly accessible, which isn't optimal. Not only is the `Token` class one more class taking up space in the designers that list classes, but it isn't designed to be generally useful. It's therefore helpful to make `Token` a nested class, which will allow it to be declared with `private` accessibility, hiding it from all classes except `Parser`.

Here's the revised code:

```
public class Parser
{
    Token[]    tokens;
    private class Token
    {
        string name;
    }
}
```

Now, nobody else can see `Token`. Another option is to make `Token` an `internal` class so it won't be visible outside the assembly, but with that solution, it is still visible inside the assembly.

Making `Token` an internal class also misses out on an important benefit of using a nested class. A nested class makes it clear to those reading the source code that the `Token` class can safely be ignored unless the internals for `Parser` are important. If this organization is applied across an entire assembly, it can help simplify the code considerably.

You can use nesting also as an organizational feature. If the `Parser` class was within a namespace named `Language`, you might require a separate namespace named `Parser` to nicely organize the classes for `Parser`. The `Parser` namespace would contain the `Token` class and a renamed `Parser` class. By using nested classes, the `Parser` class could be left in the `Language` namespace and contain the `Token` class.

Other Nesting

Classes aren't the only types that can be nested; interfaces, structs, delegates, and enums can also be nested within a class.

Creation, Initialization, Destruction

In any object-oriented system, dealing with the creation, initialization, and destruction of objects is important. In the .NET runtime, the programmer can't control the destruction of objects, but it's helpful to know the other areas that can be controlled.

Constructors

If there are no default constructors, the C# compiler will create a public parameterless constructor.

A constructor can invoke a constructor of the base type by using the `base` syntax, like this:

```
using System;
public class BaseClass
{
    public BaseClass(int x)
    {
        this.x = x;
    }
```

```
    public int X
    {
        get
        {
            return(x);
        }
    }
    int    x;
}
public class Derived: BaseClass
{
    public Derived(int x): base(x)
    {
    }
}
class Test
{
    public static void Main()
    {
        Derived d = new Derived(15);
        Console.WriteLine("X = {0}", d.X);
    }
}
```

In this example, the constructor for the `Derived` class merely forwards the construction of the object to the `BaseClass` constructor.

Sometimes it's useful for a constructor to forward to another constructor in the same object, as in the following example:

```
using System;
class MyObject
{
    public MyObject(int x)
    {
        this.x = x;
    }
    public MyObject(int x, int y): this(x)
    {
        this.y = y;
    }
    public int X
    {
        get
        {
            return(x);
        }
    }
```

```
    public int Y
    {
        get
        {
            return(y);
        }
    }
    int x;
    int y;
}
class Test
{
    public static void Main()
    {
        MyObject my = new MyObject(10, 20);
        Console.WriteLine("x = {0}, y = {1}", my.X, my.Y);
    }
}
```

Private constructors are—not surprisingly—usable only from within the class on which they're declared. If the only constructor on the class is private, this prevents any user from instantiating an instance of the class, which is useful for classes that are merely containers of static functions (such as `System.Math`, for example). C# 2.0 adds a new feature to accomplish a similar task; see the "Static Classes" section later in this chapter.

Private constructors are also used to implement the singleton pattern, when there should be only a single instance of a class within a program. This is usually done as follows:

```
public class SystemInfo
{
private static SystemInfo cache = null;
    private static object cacheLock = new object();

    private SystemInfo()
    {
        // useful stuff here…
    }

    public static SystemInfo GetSystemInfo()
    {
        lock(cacheLock)
        {
            if (cache == null)
                cache = new SystemInfo();

            return(cache);
        }
    }
}
```

This example uses locking to make sure the code works correctly in a multithreaded environment. For more information on locking, see Chapter 31.

Initialization

If the default value of the field isn't what's desired, it can be set in the constructor. If there are multiple constructors for the object, it may be more convenient—and less error-prone—to set the value through an initializer rather than setting it in every constructor.

Here's an example of how initialization works:

```
public class Parser        // Support class
{
    public Parser(int number)
    {
        this.number = number;
    }
    int number;
}
class MyClass
{
    public int counter = 100;
    public string heading = "Top";
    private Parser parser = new Parser(100);
}
```

This is pretty convenient; the initial values can be set when a member is declared. It also makes class maintenance easier since it's clearer what the initial value of a member is.

To implement this, the compiler adds code to initialize these functions to the beginning of every constructor.

■ **Tip** As a general rule, if a member has differing values depending on the constructor used, the field value should be set in the constructor. If the value is set in the initializer, it may not be clear that the member may have a different value after a constructor call.

Destructors

Strictly speaking, C# doesn't have destructors, at least not in the way most developers think of destructors, where the destructor is called when the object is deleted.

What's known as a *destructor* in C# is known as a *finalizer* in some other languages and is called by the garbage collector when an object is collected. The programmer doesn't have direct control over when the destructor is called, and it's therefore less useful than in languages such as C++. Cleanup in a destructor is a last resort, and there should always be another method that performs the same operation so that the user can control the process directly.

When a destructor is written in C#, the compiler will automatically add a call to the base class's finalizer (if present).

For more information on this, see Chapter 38. If garbage collection is new to you, you'll probably want to read that chapter before delving into the following section.

Managing Nonmemory Resources

The garbage collector does a good job of managing memory resources, but it doesn't know anything about other resources, such as database handles, graphics handles, and so on. Because of this, classes that hold such resources will have to do the management themselves.

In many cases, this isn't a real problem; all that it takes is writing a destructor for the class that cleans up the resource, like so:

```
using System;
using System.Runtime.InteropServices;

class ResourceWrapper
{
    int handle = 0;

    public ResourceWrapper()
    {
        handle = GetWindowsResource();
    }

    ~ResourceWrapper()
    {
        FreeWindowsResource(handle);
        handle = 0;
    }

    [DllImport("dll.dll")]
    static extern int GetWindowsResource();

    [DllImport("dll.dll")]
    static extern void FreeWindowsResource(int handle);
}
```

Some resources, however, are scarce and need to be cleaned up in a more timely manner than the next time a garbage collection occurs. Since there's no way to call finalizers automatically when an object is no longer needed,[1] it needs to be done manually.

In the .NET Framework, objects can indicate that they hold on to such resources by implementing the `IDisposable` interface, which has a single member named `Dispose()`. This member does the same cleanup as the finalizer, but it also needs to do some additional work. If either its base class or any of the other resources it holds implement `IDisposable`, it needs to call `Dispose()` on them so they also get cleaned up at this time.[2] After it does this, it calls

1. The discussion why this isn't possible is long and involved. In summary, lots of really smart people tried to make it work and couldn't.
2. This is different from the finalizer. Finalizers are responsible only for their own resources, while `Dispose()` also deals with referenced resources.

GC.SuppressFinalize() so that the garbage collector won't bother to finalize this object. Here's the modified code:

```
using System;
using System.Runtime.InteropServices;

class ResourceWrapper: IDisposable
{
int handle = 0;
bool disposed = false;

    public ResourceWrapper()
    {
        handle = GetWindowsResource();
    }

    private void Dispose(bool disposing)
    {
        if(!this.disposed)
        {
            if(disposing)
            {
                // call Dispose() on our base class (if necessary) and
                // on any other resources we hold that implement IDisposable
            }
            FreeWindowsResource(handle);
            handle = 0;
        }
        disposed = true;
    }

    // does cleanup for this object only
    ~ResourceWrapper()
    {
        Dispose(false);
    }

        // dispose cleans up its object, and any objects it holds
        // that also implement IDisposable.
    public void Dispose()
    {
        Dispose(true);
        GC.SuppressFinalize(this);
    }
```

```
    [DllImport("dll.dll")]
    static extern int GetWindowsResource();

    [DllImport("dll.dll")]
    static extern void FreeWindowsResource(int handle);
}
```

If your object has semantics where another name is more appropriate than `Dispose()` (a file would have `Close()`, for example), then you should implement `IDisposable` using explicit interface implementation. You would then have the better-named function forward to `Dispose()`.

IDisposable and the Using Statement

When using classes that implement `IDisposable`, it's important to make sure `Dispose()` gets called at the appropriate time. When a class is used locally, this is easily done by wrapping the usage in `try-finally`, like in this example:

```
ResourceWrapper rw = new ResourceWrapper();
try
{
    // use rw here
}
finally
{
    if (rw != null)
        ((IDisposable) rw).Dispose();
}
```

The cast of the `rw` to `IDisposable` is required because `ResourceWrapper` could have implemented `Dispose()` with explicit interface implementation.[3] The `try-finally` is a bit ugly to write and remember, so C# provides the `using` statement to simplify the code, like this:

```
using (ResourceWrapper rw = new ResourceWrapper())
{
    // use rw here
}
```

If two or more instances of a single class are used, the `using` statement can be written like so:

```
using (ResourceWrapper rw = new ResourceWrapper(), rw2 = new ResourceWrapper())
```

For different classes, two `using` statements can be placed next to each other:

```
using (ResourceWrapper rw = new ResourceWrapper())
using (FileWrapper fw = new FileWrapper())
```

In either case, the compiler will generate the appropriate nested `try-finally` blocks.

3. See Chapter 10.

IDisposable and Longer-Lived Objects

The `using` statement provides a nice way to deal with objects that are around only for a single function. For longer-lived objects, however, there's no automatic way to make sure `Dispose()` is called.

It's fairly easy to track this through the finalizer, however. If it's important that `Dispose()` is always called, it's possible to add some error checking to the finalizer to track any such cases. You could do this with a few changes to the `ResourceWrapper` class:

```
static int finalizeCount = 0;
~ResourceWrapper()
{
    finalizeCount++;
    DoDispose();
}

 [Conditional("DEBUG")]
static void CheckDisposeUsage(string location)
{
    GC.Collect();
    GC.WaitForPendingFinalizers();
    if (finalizeCount != 0)
    {
        finalizeCount = 0;
        throw new Exception("ResourceWrapper(" + location +
        ": Dispose()=" + finalizeCount);
    }
}
```

The finalizer increments a counter whenever it's called, and the `CheckDisposeUsage()` routine first makes sure all objects are finalized and then checks to see if there were any finalizations since the last check. If so, it throws an exception.[4]

Static Fields

It's sometimes useful to define members of an object that aren't associated with a specific instance of the class but rather with the class as a whole. Such members are known as `static` members.

A *static field* is the simplest type of static member; to declare a static field, simply place the `static` modifier in front of the variable declaration. For example, you could use the following to track the number of instances of a class that were created:

4. It might make more sense to log this to a file, depending on the application.

```
using System;
class MyClass
{
    public MyClass()
    {
        instanceCount++;
    }
    public static int instanceCount = 0;
}
class Test
{
    public static void Main()
    {
        MyClass my = new MyClass();
        Console.WriteLine(MyClass.instanceCount);
        MyClass my2 = new MyClass();
        Console.WriteLine(MyClass.instanceCount);
    }
}
```

The constructor for the object increments the instance count, and the instance count can be referenced to determine how many instances of the object have been created. A static field is accessed through the name of the class rather than through the instance of the class; this is true for all static members.

■ **Note** This is unlike the C++ behavior where a static member can be accessed through either the class name or the instance name. In C++, this leads to some readability problems, as it's sometimes not clear from the code whether an access is static through an instance.

Static Member Functions

The previous example exposes an internal field, which is usually something to be avoided. You can restructure it to use a static member function instead of a static field, like in the following example:

```
using System;
class MyClass
{
    public MyClass()
    {
        instanceCount++;
    }
```

```
    public static int GetInstanceCount()
    {
        return(instanceCount);
    }
    static int instanceCount = 0;
}
class Test
{
    public static void Main()
    {
        MyClass my = new MyClass();
        Console.WriteLine(MyClass.GetInstanceCount());
    }
}
```

This now uses a static member function and no longer exposes the field to users of the class, which increases future flexibility. Because it's a static member function, it's called using the name of the class rather than the name of an instance of the class.

In the real world, this example would probably be better written using a static property, which is discussed Chapter 19.

Static Constructors

Just as there can be other static members, there can also be static constructors. A static constructor will be called before the first instance of an object is created. It's useful to do setup work that needs to be done only once.

■ **Note** Like a lot of other things in the .NET runtime world, the user has no control over when the static constructor is called; the runtime guarantees only that it's called sometime after the start of the program and before the first instance of an object is created. Therefore, it can't be determined in the static constructor that an instance is about to be created.

You can declare a static constructor simply by adding the `static` modifier in front of the constructor definition. A static constructor can't have any parameters:

```
using System;
class MyClass
{
    static MyClass()
    {
        Console.WriteLine("MyClass is initializing");
    }
}
```

There's no static analog of a destructor.

Constants

C# allows values to be defined as constants. For a value to be a constant, its value must be something that can be written as a constant. This limits the types of constants to the built-in types that can be written as literal values.

Not surprisingly, putting `const` in front of a variable means its value can't be changed. Here's an example of some constants:

```
using System;
enum MyEnum
{
    Jet
}
class LotsOLiterals
{
        // const items can't be changed.
        // const implies static.
    public const int value1 = 33;
    public const string value2 = "Hello";
    public const MyEnum value3 = MyEnum.Jet;
}
class Test
{
    public static void Main()
    {
        Console.WriteLine("{0} {1} {2}",
                LotsOLiterals.value1,
                LotsOLiterals.value2,
                LotsOLiterals.value3);
    }
}
```

Read-Only Fields

Because of the restriction on constant types being knowable at compile time, `const` can't be used in many situations.

In a `Color` class, it can be useful to have constants as part of the class for the common colors. If there were no restrictions on `const`, the following would work:

```
// error
class Color
{
    public Color(int red, int green, int blue)
    {
        this.red = red;
        this.green = green;
        this.blue = blue;
    }
```

```
    int red;
    int green;
    int blue;
        // call to new can't be used with static
    public const Color Red = new Color(255, 0, 0);
    public const Color Green = new Color(0, 255, 0);
    public const Color Blue = new Color(0, 0, 255);
}
class Test
{
    static void Main()
    {
        Color background = Color.Red;
    }
}
```

This clearly doesn't work because the static members `Red`, `Green`, and `Blue` can't be calculated at compile time. But making them normal public members doesn't work either; anybody could change red to olive drab or puce.

The `readonly` modifier is designed for exactly that situation. By applying `readonly`, you can set the value in the constructor or in an initializer, but you can't modify it later.

Because the color values belong to the class and not a specific instance of the class, they'll be initialized in the static constructor, like so:

```
class Color
{
    public Color(int red, int green, int blue)
    {
        this.red = red;
        this.green = green;
        this.blue = blue;
    }

    int red;
    int green;
    int blue;

    public static readonly Color Red;
    public static readonly Color Green;
    public static readonly Color Blue;

        // static constructor
    static Color()
    {
        Red = new Color(255, 0, 0);
        Green = new Color(0, 255, 0);
        Blue = new Color(0, 0, 255);
    }
}
```

```
class Test
{
    static void Main()
    {
        Color background = Color.Red;
    }
}
```

This provides the correct behavior.

If the number of static members was high or creating them was expensive (either in time or in memory), it might make more sense to declare them as `readonly` properties so that members could be constructed on the fly as needed.

On the other hand, it might be easier to define an enumeration with the different color names and return instances of the values as needed:

```
class Color
{
    public Color(int red, int green, int blue)
    {
        this.red = red;
        this.green = green;
        this.blue = blue;
    }

    public enum PredefinedEnum
    {
        Red,
        Blue,
        Green
    }
    public static Color GetPredefinedColor(
    PredefinedEnum pre)
    {
        switch (pre)
        {
            case PredefinedEnum.Red:
                return(new Color(255, 0, 0));

            case PredefinedEnum.Green:
                return(new Color(0, 255, 0));

            case PredefinedEnum.Blue:
                return(new Color(0, 0, 255));

            default:
                return(new Color(0, 0, 0));
        }
    }
```

```
    int red;
    int blue;
    int green;
}
class Test
{
    static void Main()
    {
        Color background =
            Color.GetPredefinedColor(Color.PredefinedEnum.Blue);
    }
}
```

This requires a little more typing, but there isn't a startup penalty or lots of objects taking up space. It also keeps the class interface simple; if there were 30 members for predefined colors, the class would be much harder to understand.[5]

■ **Note** Experienced C++ programmers are probably cringing at the last code example. It embodies one of the classic problems with the way C++ deals with memory management. Passing back an allocated object means the caller has to free it. It's pretty easy for the user of the class to either forget to free the object or lose the pointer to the object, which leads to a memory leak.

In C#, however, this isn't an issue; the runtime handles memory allocation. In the preceding example, the object created in the `Color.GetPredefinedColor()` function gets copied immediately to the background variable and then is available for collection when the background goes out of scope.

Static Classes

New to C# 2.0 are static classes. At times, all the methods (and potentially, properties) on a class will be static. This typically happens in utility classes, such as math libraries, where saving state in instance variables serves no benefit. As mentioned earlier, it's good programming to make the constructor private for classes like this, as it prevents users of the class from accidentally instantiating the class just to use a static method (C# prevents this, but other languages may not). However, the private constructor is merely a convention and doesn't carry any compiler enforcement.

This lack of compiler enforcement has two problems: one, a developer can forget to add this feature to the class, and two, there's nothing preventing the developer from accidentally adding an instance method. These problems aren't contrived or mere thought experiments. `System.Environment` offers a number of static methods and properties that allow the state of the computing environment to be queried. As there's only a single computing environment from the point of view of a process, all members are static. However, the static modifier on the

5. For an explanation on why a `default` case is required, see Chapter 21.

`HasShutdownStarted` property was accidentally omitted in version 1.0 of the .NET Framework library, and the product shipped without the ability to access this property.

The solution to the problem is quite simple: classes can be marked as static, which prevents them from having constructors, destructors, instance methods, and instance member variables. To declare a class as static, place the `static` modifier before the `class` keyword:

```
static class Util {
 static int PerformCal() { return 0; }
}
```

With the static keyword in place, the compiler will enforce that no instance members are present and will prevent a client of the class from unwittingly constructing an instance of the static class.

Partial Classes

Despite being one of the "big four" new features of C# 2.0 (along with generics, iterators, and anonymous methods), partial classes are relatively uninteresting. They simply allow a class to be split over multiple source files and have no effect on the intermediate code produced by the compiler. The motivation for partial classes is as follows:

- Some classes grow too big to be comfortably worked on within the editor using a single file.
- Separate developers may want to work on the same class simultaneously without using the multiple checkout facilities of a source-control system.
- Development environments may want to maintain a split between human-generated and machine-generated code.

Despite their mundane nature, these problems are genuine concerns when they do occur and where sufficiently important to warrant an extension to the C# language.

You implement partial classes by placing the `partial` modifier before the `class` keyword:

```
//declare once
partial class MyPartialClass { }

//and then again - this could be in a separate source file
partial class MyPartialClass { }
```

The modifier simply informs the compiler that fragments of the class may exist in multiple source files. It's legal for a partial class to exist entirely within the one source file, or it can be spread out over any number of source files. Using a consistent naming convention to allow all the fragments of a class to be located is recommended, but not enforced, by the compiler. It will generally be worth using development environment features such as Visual Studio's Class View to provide a holistic view of a class if the partial class modifier is used.

The following are other points to keep in mind:

- Structs can use the `partial` modifier, but enumerations can't.
- Assemblies form the boundaries of classes, and a partial class can't span multiple assemblies.
- If one file marks a class as partial, all other declarations of the class must also use the `partial` modifier.
- It's legal for each partial class to add features to the class as long as they aren't mutually exclusive.

For example, it's a compiler error to define different base types or to have two implementations of the same method. Besides these nonsensical scenarios, the programmer is free to make the job of the maintenance programmer as difficult or easy as possible. The following code will compile and run without any errors:

```
//in file1.cs
partial class MyPartialClass : MyBase
{
 public void Dispose() { }
}

//in file2.cs
public class MyBase{}
partial class MyPartialClass : IDisposable { }
```

In the interests of comprehensibility and maintainability, code that uses partial types should apply all class modifiers and nominate all base classes and interfaces in a main file and then group all logically similar functionality in a single file. Of course, you should avoid partial classes if they aren't needed, and you shouldn't employ them gratuitously.

CHAPTER 9

Structs (Value Types)

You'll use classes to implement most objects. Sometimes, however, it may be desirable to create an object that behaves like one of the built-in types—one that's cheap and fast to allocate and doesn't have the overhead of references. In that case, you can use a value type by declaring a struct in C#.

Structs act similarly to classes but with a few added restrictions. They can't inherit from any other type (though they implicitly inherit from `object`), and other classes can't inherit from them.[1]

A Point Struct

In a graphics system, a value class could encapsulate a point. Here's how you'd declare it:

```
using System;
struct Point
{
    public Point(int x, int y)
    {
        this.x = x;
        this.y = y;
    }
    public override string ToString()
    {
        return(String.Format("({0}, {1})", x, y));
    }

    public int x;
    public int y;
}
```

1. Technically, structs are derived from `System.ValueType`, but that's only an implementation detail. From a language perspective, they act as if they're derived from `System.Object`.

```
class Test
{
    public static void Main()
    {
        Point    start = new Point(5, 5);
        Console.WriteLine("Start: {0}", start);
    }
}
```

The x and y components of the Point can be accessed. In the Main() function, a Point is created using the new keyword. For value types, new creates an object on the stack and then calls the appropriate constructor.

The call to Console.WriteLine() is a bit mysterious. If Point is allocated on the stack, how does that call work?

Boxing and Unboxing

In C# and the .NET runtime world, a little bit of magic takes place to make value types look like reference types, and that magic is called *boxing*. As magic goes, it's pretty simple. In the call to Console.WriteLine(), the compiler is looking for a way to convert start to an object because the type of the second parameter to WriteLine() is object. For a reference type (in other words, a class), this is easy because object is the base class of all classes. The compiler merely passes an object reference that refers to the class instance.

There's no reference-based instance for a value class, however, so the C# compiler allocates a reference-type "box" for the Point, marks the box as containing a Point, and copies the value of the Point into the box. It's now a reference type, and you can treat it like an object.

This reference is then passed to the WriteLine() function, which calls the ToString() function on the boxed Point, which gets dispatched to the ToString() function, and the code writes the following:

```
Start: (5, 5)
```

Boxing happens automatically whenever a value type is used in a location that requires (or could use) an object.

The boxed value is retrieved into a value type by unboxing it:

```
int v = 123;
object o = v;        // box the int 123
int v2 = (int) o;    // unbox it back to an integer
```

Assigning the object o, the value 123 boxes the integer, which is then extracted back on the next line. That cast to int is required because the object o could be any type of object and because the cast could fail.

Figure 9-1 shows how this would be represented. Assigning the int to the object variable results in the box being allocated on the heap and the value being copied into the box. The box is then labeled with the type it contains so the runtime knows the type of the boxed object.

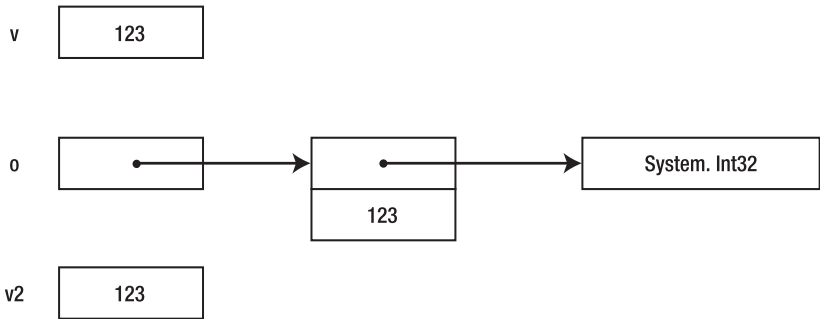

Figure 9-1. *Boxing and unboxing a value type*

During the unboxing conversion, the type must match exactly; a boxed value type can't be unboxed to a compatible type:

```
object o = 15;
short s = (short) o;        // fails, o doesn't contain a short
short t = (short)(int) o;   // this works
```

It's fairly rare to write code that does boxing explicitly. It's much more common to write code where the boxing happens because the value type is passed to a function that expects a parameter of type `object`, like the following code:

```
int value = 15;
DateTime date = new DateTime();
Console.WriteLine("Value, Date: {0} {1}", value, date);
```

In this case, both `value` and `date` will be boxed when `WriteLine()` is called.

Structs and Constructors

Structs and constructors behave a bit differently than classes. In classes, an instance must be created by calling `new` before the object is used; if `new` isn't called, there will be no created instance, and the reference will be null.

There's no reference associated with a struct, however. If `new` isn't called on the struct, an instance that has all of its fields zeroed is created. In some cases, a user can then use the instance without further initialization. For example:

```
using System;
struct Point
{
    int x;
    int y;
```

```
    Point(int x, int y)
    {
        this.x = x;
        this.y = y;
    }
    public override string ToString()
    {
        return(String.Format("({0}, {1})", x, y));
    }
}
class Test
{
    public static void Main()
    {
        Point[] points = new Point[5];
        Console.WriteLine("[2] = {0}", points[2]);
    }
}
```

Although this struct has no default constructor, it's still easy to get an instance that didn't come through the right constructor.

It's therefore important to make sure that the all-zeroed state is a valid initial state for all value types.

A default (parameterless) constructor for a struct could set different values than the all-zeroed state, which would be unexpected behavior. The .NET runtime therefore prohibits default constructors for structs.

Design Guidelines

You should use structs only for types that are really just pieces of data—for types that could be used in a similar way to the built-in types. A type, for example, is like the built-in type `decimal`, which is implemented as a value type.

Even if more complex types *can* be implemented as value types, they probably shouldn't be, since the value type semantics will probably not be expected by the user. The user will expect that a variable of the type could be `null`, which isn't possible with value types.

Immutable Classes

Value types nicely result in value semantics, which is great for types that "feel like data." But what if it's a type that needs to be a class type for implementation reasons but is still a data type, such as the `string` type?

To get a class to behave as if it were a value type, you need to write the class as an immutable type. Basically, an immutable type is one designed so that it's not possible to tell it has reference semantics for assignment.

Consider the following example, with `string` written as a normal class:

```
string s = "Hello There";
string s2 = s;
s.Replace("Hello", "Goodbye");
```

Because `string` is a reference type, both `s` and `s2` will end up referring to the same string instance. When that instance is modified through `s`, the views through both variables will be changed.

The way to get around this problem is simply to prohibit any member functions that change the value of the class instance. In the case of `string`, member functions that look like they'd change the value of the string instead return a new string with the modified value.

A class where there are no member functions that can change—or *mutate*—the value of an instance is called an *immutable* class. The revised example looks like this:

```
string s = "Hello There";
string s2 = s;
s = s.Replace("Hello", "Goodbye");
```

After the third line has been executed, `s2` still points to the original instance of the string, and `s` now points to a new instance containing the modified value.

CHAPTER 10

Interfaces

Interfaces are closely related to abstract classes that have all members abstract.

A Simple Example

The following code defines the interface `IScalable` and the class `TextObject`, which implements the interface, meaning that it contains implementations of all the functions defined in the interface:

```
public class DiagramObject
{
    public DiagramObject() {}
}

interface IScalable
{
    void ScaleX(float factor);
    void ScaleY(float factor);
}
    // A diagram object that also implements IScalable
public class TextObject: DiagramObject, IScalable
{
    public TextObject(string text)
    {
        this.text = text;
    }
        // implementing IScalable.ScaleX()
    public void ScaleX(float factor)
    {
        // scale the object here.
    }

        // implementing IScalable.ScaleY()
    public void ScaleY(float factor)
    {
        // scale the object here.
    }
```

```
    private string text;
}

class Test
{
    public static void Main()
    {
        TextObject text = new TextObject("Hello");

        IScalable scalable = (IScalable) text;
        scalable.ScaleX(0.5F);
        scalable.ScaleY(0.5F);
    }
}
```

This code implements a system for drawing diagrams. All the objects derive from `DiagramObject`, so they can implement common virtual functions (not shown in this example). Some of the objects can be scaled, and this is expressed by the presence of an implementation of the `IScalable` interface.

Listing the interface name with the base class name for `TextObject` indicates that `TextObject` implements the interface. This means `TextObject` must have functions that match every function in the interface. Interface members have no access modifiers, and the class members that implement the interface members must be publicly accessible.

When an object implements an interface, a reference to the interface can be obtained by casting to the interface. This can then be used to call the functions on the interface.

This example could have been done with abstract methods, by moving the `ScaleX()` and `ScaleY()` methods to `DiagramObject` and making them virtual. The "Design Guidelines" section later in this chapter discusses when to use an abstract method and when to use an interface.

Working with Interfaces

Typically, code doesn't know whether an object supports an interface, so it needs to check whether the object implements the interface before doing the cast:

```
using System;
interface IScalable
{
    void ScaleX(float factor);
    void ScaleY(float factor);
}
public class DiagramObject
{
    public DiagramObject() {}
}
public class TextObject: DiagramObject, IScalable
{
    public TextObject(string text)
```

```
    {
        this.text = text;
    }
        // implementing ISclalable.ScaleX()
    public void ScaleX(float factor)
    {
        Console.WriteLine("ScaleX: {0} {1}", text, factor);
        // scale the object here.
    }

        // implementing IScalable.ScaleY()
    public void ScaleY(float factor)
    {
        Console.WriteLine("ScaleY: {0} {1}", text, factor);
        // scale the object here.
    }

    private string text;
}
class Test
{
    public static void Main()
    {
        DiagramObject[] dArray = new DiagramObject[100];

        dArray[0] = new DiagramObject();
        dArray[1] = new TextObject("Text Dude");
        dArray[2] = new TextObject("Text Backup");

        // array gets initialized here, with classes that
        // derive from DiagramObject. Some of them implement
        // IScalable.

        foreach (DiagramObject d in dArray)
        {
            if (d is IScalable)
            {
                IScalable scalable = (IScalable) d;
                scalable.ScaleX(0.1F);
                scalable.ScaleY(10.0F);
            }
        }
    }
}
```

Before the cast is done, it's checked to make sure the cast will succeed. If it will succeed, the object is cast to the interface, and the scale functions are called.

This construct unfortunately checks the type of the object twice: once as part of the `is` operator and once as part of the cast. This is wasteful, since the cast can never fail.

One way around this is to restructure the code with exception handling, but that's not a great idea because it'd make the code more complex, and exception handling should generally be reserved for exceptional conditions. It's also not clear whether it'd be faster, since exception handling has some overhead.

The as Operator

C# provides a special operator for this situation, the `as` operator. Using the `as` operator, you can rewrite the loop as follows:

```
using System;
interface IScalable
{
    void ScaleX(float factor);
    void ScaleY(float factor);
}
public class DiagramObject
{
    public DiagramObject() {}
}
public class TextObject: DiagramObject, IScalable
{
    public TextObject(string text)
    {
        this.text = text;
    }
        // implementing IScalable.ScaleX()
    public void ScaleX(float factor)
    {
        Console.WriteLine("ScaleX: {0} {1}", text, factor);
        // scale the object here.
    }

        // implementing IScalable.ScaleY()
    public void ScaleY(float factor)
    {
        Console.WriteLine("ScaleY: {0} {1}", text, factor);
        // scale the object here.
    }

    private string text;
}
class Test
{
```

```
    public static void Main()
    {
        DiagramObject[] dArray = new DiagramObject[100];

        dArray[0] = new DiagramObject();
        dArray[1] = new TextObject("Text Dude");
        dArray[2] = new TextObject("Text Backup");

        // array gets initialized here, with classes that
        // derive from DiagramObject. Some of them implement
        // IScalable.

        foreach (DiagramObject d in dArray)
        {
            IScalable scalable = d as IScalable;
            if (scalable != null)
            {
                scalable.ScaleX(0.1F);
                scalable.ScaleY(10.0F);
            }
        }
    }
}
```

The as operator checks the type of the left operand, and if it can be converted explicitly to the right operand, the result of the operator is the object converted to the right operand. If the conversion will fail, the operator returns null.

Both the is and as operators can also be used with classes.

Interfaces and Inheritance

When converting from an object to an interface, the inheritance hierarchy is searched until it finds a class that lists the interface on its base list. Having the right functions alone isn't enough. For example:

```
using System;
interface IHelper
{
    void HelpMeNow();
}
public class Base: IHelper
{
    public void HelpMeNow()
    {
        Console.WriteLine("Base.HelpMeNow()");
    }
}
```

```
    // Does not implement IHelper, though it has the right
    // form.
public class Derived: Base
{
    public new void HelpMeNow()
    {
        Console.WriteLine("Derived.HelpMeNow()");
    }
}
class Test
{
    public static void Main()
    {
        Derived der = new Derived();
        der.HelpMeNow();
        IHelper helper = (IHelper) der;
        helper.HelpMeNow();
    }
}
```

This code gives the following output:

```
Derived.HelpMeNow()
Base.HelpMeNow()
```

It doesn't call the `Derived` version of `HelpMeNow()` when calling through the interface, even though `Derived` does have a function of the correct form, because `Derived` doesn't implement the interface.

Design Guidelines

Both interfaces and abstract classes have similar behaviors and can be used in similar situations. Because of how they work, however, interfaces make sense in some situations and abstract classes in others. This section contains a few guidelines to determine whether a capability should be expressed as an interface or an abstract class.

The first thing to check is whether the object would be properly expressed using the "is-a" relationship. In other words, is the capability an object, and would the derived classes be examples of that object?

Another way of looking at this is to list what kind of objects would want to use this capability. If the capability would be useful across a range of different objects that aren't really related to each other, an interface is the proper choice.

■ **Caution** Because there can be only one base class in the .NET runtime world, this decision is pretty important. If a base class is required, users will be disappointed if they already have a base class and are unable to use the feature.

When using interfaces, remember that there's no versioning support for an interface. If a function is added to an interface after users are already using it, their code will break at runtime and their classes won't properly implement the interface until the appropriate modifications are made.

Multiple Implementation

Unlike object inheritance, a class can implement more than one interface:

```
interface IFoo
{
    void ExecuteFoo();
}

interface IBar
{
    void ExecuteBar();
}

class Tester: IFoo, IBar
{
    public void ExecuteFoo() {}
    public void ExecuteBar() {}
}
```

This works fine if there are no name collisions between the functions in the interfaces. But if the example is just a bit different, there might be a problem:

```
// error
interface IFoo
{
    void Execute();
}

interface IBar
{
    void Execute();
}

class Tester: IFoo, IBar
{
        // IFoo or IBar implementation?
    public void Execute() {}
}
```

Does `Tester.Execute()` implement `IFoo.Execute()` or `IBar.Execute()`?

In this example, `IFoo.Execute()` and `IBar.Execute()` are implemented by the same function. If they're supposed to be separate, one of the member names could be changed, but that's not a good solution in most cases.

More seriously, if `IFoo` and `IBar` came from different vendors, they couldn't be changed.

The .NET runtime and C# support a technique known as *explicit interface implementation*, which allows a function to specify which interface member it's implementing.

Explicit Interface Implementation

To specify which interface a member function is implementing, qualify the member function by putting the interface name in front of the member name.

Here's the previous example, revised to use explicit interface implementation:

```
using System;
interface IFoo
{
    void Execute();
}

interface IBar
{
    void Execute();
}

class Tester: IFoo, IBar
{
    void IFoo.Execute()
    {
        Console.WriteLine("IFoo.Execute implementation");
    }
    void IBar.Execute()
    {
        Console.WriteLine("IBar.Execute implementation");
    }
}

class Test
{
    public static void Main()
    {
        Tester tester = new Tester();

        IFoo iFoo = (IFoo) tester;
        iFoo.Execute();

        IBar iBar = (IBar) tester;
        iBar.Execute();
    }
}
```

This prints the following:

```
IFoo.Execute implementation
IBar.Execute implementation
```

This is what you expected. But what does the following test class do?

```
// error
using System;
interface IFoo
{
    void Execute();
}

interface IBar
{
    void Execute();
}

class Tester: IFoo, IBar
{
    void IFoo.Execute()
    {
        Console.WriteLine("IFoo.Execute implementation");
    }
    void IBar.Execute()
    {
        Console.WriteLine("IBar.Execute implementation");
    }
}
class Test
{
    public static void Main()
    {
        Tester tester = new Tester();

        tester.Execute();
    }
}
```

Is `IFoo.Execute()` called, or is `IBar.Execute()` called?

The answer is that neither is called. There's no access modifier on the implementations of `IFoo.Execute()` and `IBar.Execute()` in the `Tester` class, and therefore the functions are private and can't be called.

In this case, this behavior isn't because the public modifier wasn't used on the function; it's because access modifiers are prohibited on explicit interface implementations, so the only way the interface can be accessed is by casting the object to the appropriate interface.

To expose one of the functions, add a forwarding function to `Tester`:

```
using System;
interface IFoo
{
    void Execute();
}

interface IBar
{
    void Execute();
}
class Tester: IFoo, IBar
{
    void IFoo.Execute()
    {
        Console.WriteLine("IFoo.Execute implementation");
    }
    void IBar.Execute()
    {
        Console.WriteLine("IBar.Execute implementation");
    }

    public void Execute()
    {
        ((IFoo) this).Execute();
    }
}
class Test
{
    public static void Main()
    {
        Tester tester = new Tester();

        tester.Execute();
    }
}
```

Now, calling the `Execute()` function on an instance of `Tester` will forward to `Tester.IFoo.Execute()`.

You can use this hiding for other purposes, as detailed in the next section.

Implementation Hiding

Sometimes it makes sense to hide the implementation of an interface from the users of a class, either because it's not generally useful or because you just want to reduce the member clutter. Doing so can make an object much easier to use. For example:

```
using System;
class DrawingSurface
{

}
interface IRenderIcon
{
    void DrawIcon(DrawingSurface surface, int x, int y);
    void DragIcon(DrawingSurface surface, int x, int y, int x2, int y2);
    void ResizeIcon(DrawingSurface surface, int xsize, int ysize);
}
class Employee: IRenderIcon
{
    public Employee(int id, string name)
    {
        this.id = id;
        this.name = name;
    }
    void IRenderIcon.DrawIcon(DrawingSurface surface, int x, int y)
    {
    }
    void IRenderIcon.DragIcon(DrawingSurface surface, int x, int y,
                                                       int x2, int y2)
    {
    }
    void IRenderIcon.ResizeIcon(DrawingSurface surface, int xsize, int ysize)
    {
    }
    int id;
    string name;
}
```

If the interface had been implemented normally, the `DrawIcon()`, `DragIcon()`, and `ResizeIcon()` member functions would be visible as part of `Employee`, which might be confusing to users of the class. By implementing them through explicit implementation, they can be accessed only through the interface.

Interfaces Based on Interfaces

You can combine interfaces to form new interfaces. The `ISortable` and `ISerializable` interfaces can be combined, and new interface members can be added:

```
using System.Runtime.Serialization;
using System;
interface IComparableSerializable :
    IComparable, ISerializable
{
    string GetStatusString();
}
```

A class that implements `IComparableSerializable` would need to implement all the members in `IComparable`, `ISerializable`, and the `GetStatusString()` function introduced in `IComparableSerializable`.

Interfaces and Structs

Like classes, structs can also implement interfaces. Here's a short example:

```
using System;
struct Number: IComparable
{
    int value;

    public Number(int value)
    {
        this.value = value;
    }
    public int CompareTo(object obj2)
    {
        Number num2 = (Number) obj2;
        if (value < num2.value)
            return(-1);
        else if (value > num2.value)
            return(1);
        else
            return(0);
    }
}
class Test
{
    public static void Main()
    {
        Number x = new Number(33);
        Number y = new Number(34);

        IComparable Ic = (IComparable) x;
        Console.WriteLine("x compared to y = {0}", Ic.CompareTo(y));
    }
}
```

This struct implements the `IComparable` interface, which compares the values of two elements for sorting or searching operations.

Like classes, interfaces are reference types, so there's a boxing operation involved here. When a value type is cast to an interface, the value type is boxed and the interface pointer is to the boxed value type.

CHAPTER 11

Versioning and Aliases

A software project rarely exists as a single version of code that's never revised, unless the software never sees the light of day. In most cases, the software library writer is going to want to change some things, and the client will need to adapt to such changes.

Dealing with such issues is known as *versioning*, and it's one of the hardest tasks to do in software. One reason why it's tough is that it requires a bit of planning and foresight; you have to determine the areas that might change and modify the design to allow change.

Another reason why versioning is tough is that most execution environments don't provide much help to the programmer. In C++, compiled code has internal knowledge of the size and layout of all classes burned into it. With care, you can make some revisions to the class without forcing all users to recompile, but the restrictions are fairly severe. When compatibility is broken, all users need to recompile to use the new version. This may not be that bad, but installing a new version of a library may cause other applications that use an older version of the library to cease functioning.

Although it's still possible to write code that has versioning problems, .NET makes versioning easier by deferring the physical layout of classes and members until JIT compilation time. Rather than providing physical layout data, a .NET assembly provides metadata that allows a type to be laid out and accessed in a manner that makes sense for a particular process architecture.

A Versioning Example

The following code presents a simple versioning scenario and explains why C# has `new` and `override` keywords. The program uses a class named `Control`, which is provided by another company.

```
public class Control
{
}
public class MyControl: Control
{
}
```

During implementation of `MyControl`, a developer can add the virtual function `Foo()`:

```
public class Control
{
}
```

```
public class MyControl: Control
{
    public virtual void Foo() {}
}
```

This works well, until an upgrade notice arrives from the suppliers of the `Control` object. The new library includes a virtual `Foo()` function on the `Control` object:

```
public class Control
{
        // newly added virtual
    public virtual void Foo() {}
}
public class MyControl: Control
{
    public virtual void Foo() {}
}
```

`Control` uses `Foo()` as the name of the function, but this is only a coincidence. In the C++ world, the compiler will assume that the version of `Foo()` in `MyControl` does what a virtual override of the `Foo()` in `Control` should do and will blindly call the version in `MyControl`. This is bad.

In the Java world, this will also happen, but things can be a fair bit worse; if the virtual function doesn't have the same return type, the class loader will consider the `Foo()` in `MyControl` to be an invalid override of the `Foo()` in `Control`, and the class will fail to load at runtime.

In C# and the .NET runtime, a function defined with `virtual` is always considered to be the root of a virtual dispatch. If a function is introduced into a base class that could be considered a base virtual function of an existing function, the runtime behavior remains the same. When the class is next compiled, however, the compiler will generate a warning, requesting that the programmer specify their versioning intent. Returning to the example, to use the default behavior of not considering the function an override, add the `new` modifier in front of the function:

```
class Control
{
    public virtual void Foo() {}
}
class MyControl: Control
{
        // not an override
    public new virtual void Foo() {}
}
```

The presence of `new` will suppress the warning.

If, on the other hand, the derived version is an override of the function in the base class, use the `override` modifier:

```
class Control
{
    public virtual void Foo() {}
}
```

```
class MyControl: Control
{
        // an override for Control.Foo()
    public override void Foo() {}
}
```

This tells the compiler the function really is an override.

■ **Caution** About this time, you may be thinking, "I'll just put `new` on all my virtual functions, and then I'll never have to deal with the situation again." However, doing so reduces the value that the `new` annotation has to somebody reading the code. If `new` is used only when it's required, the reader can find the base class and understand what function isn't being overridden. If `new` is used indiscriminately, the user will have to refer to the base class every time to see if `new` has meaning.

Coding for Versioning

The C# language provides some assistance in writing code that versions well:

- Methods, for example, aren't virtual by default. This helps limit the areas where versioning is constrained to those areas that were intended by the designer of the class and prevents "stray virtuals" that constrain future changes to the class.
- C# also has lookup rules designed to aid in versioning. Adding a new function with a more specific overload (in other words, one that matches a parameter better) to a base class won't prevent a less-specific function in a derived class from being called,[1] so a change to the base class won't break existing behavior.

A language can do only so much. That's why versioning is something to keep in mind when designing classes. One specific area that has some versioning trade-offs is the choice between classes and interfaces.

The choice between class and interface should be fairly straightforward. Classes are appropriate only for "is-a" relationships (where the derived class is really an instance of the base class), and interfaces are appropriate for all others. If you choose to use an interface, however, good design becomes more important because interfaces simply don't version; when a class implements an interface, it needs to implement the interface *exactly*, and adding another method later will mean that classes that thought they implemented the interface no longer do.

External Assembly Aliases

At the CLR level, a type reference is fully qualified by both a namespace and a full assembly name. This makes it possible to reference two types that have identical names and that exist in identical namespaces, with the types being differentiated by the name of their assembly. While this

1. See Chapter 7 for an example.

scenario isn't necessarily common, it can occur with in-house projects where developers have been loose with adding full namespace hierarchies or where versioning has been accomplished by using the traditional Windows style of DLL versioning that involves including the version number in the binary's filename. If you need to reference multiple versions of an assembly that has been versioned in this manner, using namespace qualifiers is insufficient to specify which version of the type should be used.

To solve this problem, C# 2.0 introduces the compile-time ability to specify an assembly alias for types so multiple types that exist in the same namespace can be used. A source code file that wants to employ this alias uses the `extern alias` statement to bring the alias into scope. This creates a new top-level namespace for the types from the aliased assembly, allowing two identically named types to be distinguished based on aliases.

To provide an example of using external assembly aliases, consider the two `Math` classes that have been developed independently at a particular company:

```
//in assembly maths.dll
namespace AcmeScientific
{
 public class Math
 {
  public int Calc() { return 0;}
 }
}

//in assembly utils.dll
namespace AcmeScientific
{
 public class Math
 {
  public void DoSums() {}
 }
}
```

If you needed to use functionality from the `Math` classes in both of these assemblies in previous versions of C#, you'd need to make changes to the type name or namespace hierarchy in one of these assemblies. To resolve this problem in C# 2.0, the assemblies that the `Math` classes live in are aliased as part of the `/reference` C# compiler option (aliases shown in *italics*):

```
csc /reference:Maths=maths.dll /reference:Utils=utils.dll /t:exe ➥
  /out:MyApp.exe source.cs
```

To use the aliases in code, add `extern alias` declarations to the top of the source code file and reference the types within the aliased assemblies using the alias name, a double colon, and then the full namespace name:

```
extern alias Maths;
extern alias Utils;

namespace AcmeScientific.MyApp
{
class App{
  static void Main()
  {
   Maths::AcmeScientific.Math m = new Maths::AcmeScientific.Math();
   m.Calc();

   Utils::AcmeScientific.Math u = new Utils::AcmeScientific.Math();
   u.DoSums();
  }
 }
}
```

Because the number of source code files that need to distinguish between types based on assembly aliases is likely to be relatively small, it's possible to use the `using` statement to bring an aliased namespace into the global namespace for that file. In the previous example, if the `Math` type in the `Maths` assembly was the only one needed in a source file, the following code would be easier than qualifying the `Math` class with the full alias and namespace every time:

```
extern alias Maths;
using Maths::AcmeScientific;

namespace AcmeScientific.MyApp
{
class App{
  static void Main()
  {
   Math m = new Math();
   m.Calc();
  }
 }
}
```

You can place a type in both the global namespace and an aliased assembly namespace by including it twice in the `/references` switch—once with an alias and once without. Using this technique, you can rewrite the previous snippet as follows:

```
//command line to compile
//csc /reference:maths.dll  /reference:Maths=maths.dll
// /reference:Utils=utils.dll /t:exe /out:MyApp.exe source.cs
using AcmeScientific;
```

```
namespace AcmeScientific.MyApp
{
class App{
  static void Main()
  {
   Math m = new Math();
   m.Calc();
  }
 }
}
```

Along the same theme, two assemblies can share the same alias as long as the types that need to be used within these assemblies can be differentiated based on their names (including the namespace). Types from assemblies that aren't aliased are implicitly placed in the global namespace and can be fully qualified by using the `global` assembly alias. Incorporating this into the previous example, it becomes the following:

```
using global::AcmeScientific;

namespace AcmeScientific.MyApp
{
class App
 {
  static void Main()
  {
   Math m = new Math();
   m.Calc();
  }
 }
}
```

The C# compiler will prevent using `global` as an assembly alias.

■ **Note** Extern aliases are available in a C# project in VS .NET 2005 through the Properties window. Select an assembly in the Solution Explorer, and then change the `Aliases` property as required. By default, an assembly will have an alias of `global`. To provide multiple aliases for the one assembly, enter a comma-delimited list of alias names.

CHAPTER 12

Statements and Flow of Execution

This chapter details the statements available within the C# language.

Selection Statements

The selection statements perform operations based on the value of an expression.

if

The `if` statement in C# requires that the condition inside the `if` statement evaluate to an expression of type `bool`. In other words, the following is illegal:

```
// error
using System;
class Test
{
    public static void Main()
    {
        int    value;

        if (value)        // invalid
            System.Console.WriteLine("true");

        if (value == 0)    // must use this
            System.Console.WriteLine("true");
    }
}
```

switch

`switch` statements have often been error-prone; it's just too easy to inadvertently omit a `break` statement at the end of a `case` or, more likely, to not notice that there's fall-through when reading code.

C# gets rid of this possibility by requiring that there be either a `break` at the end of every `case` block or a `goto` another `case` label in the `switch`. For example:

```
using System;
class Test
{
    public void Process(int i)
    {
        switch (i)
        {
            case 1:
            case 2:
                // code here handles both 1 and 2
                Console.WriteLine("Low Number");
                break;

            case 3:
                Console.WriteLine("3");
                goto case 4;

            case 4:
                Console.WriteLine("Middle Number");
                break;

            default:
                Console.WriteLine("Default Number");
                break;
        }
    }
}
```

C# also allows the `switch` statement to be used with string variables:

```
using System;
class Test
{
    public void Process(string htmlTag)
    {
        switch (htmlTag)
        {
            case "P":
                Console.WriteLine("Paragraph start");
                break;
            case "DIV":
                Console.WriteLine("Division");
                break;
```

```
            case "FORM":
                Console.WriteLine("Form Tag");
                break;
            default:
                Console.WriteLine("Unrecognized tag");
                break;
        }
    }
}
```

Not only is it easier to write a `switch` statement than a series of `if` statements but it's also more efficient, as the compiler uses an efficient algorithm to perform the comparison.

For small numbers of entries[1] in the `switch`, the compiler uses a feature in the .NET runtime known as *string interning*. The runtime maintains an internal table of all constant strings so that all occurrences of that string in a single program will have the same object. In the `switch`, the compiler looks up the `switch` string in the runtime table. If it isn't there, the string can't be one of the cases, so the `default` case is called. If it's found, a sequential search is done of the interned `case` strings to find a match.

For larger numbers of entries in the `case`, the compiler generates a hash function and hash table and uses the hash table to efficiently look up the string.[2]

Iteration Statements

Iteration statements are often known as *looping statements*, and they perform operations while a specific condition is true.

while

The `while` loop functions as expected: while the condition is true, the loop is executed. Like the `if` statement, the `while` requires a Boolean condition:

```
using System;
class Test
{
    public static void Main()
    {
        int n = 0;
        while (n < 10)
        {
            Console.WriteLine("Number is {0}", n);
            n++;
        }
    }
}
```

1. The actual number is determined based upon the performance trade-offs of each method.
2. If you're unfamiliar with hashing, consider looking at the `System.Collections.HashTable` class or a good algorithms book.

You can use the `break` statement to exit the `while` loop, and you can use the `continue` statement to skip to the closing brace of the `while` block for this iteration and then continue with the next iteration:

```
using System;
class Test
{
    public static void Main()
    {
        int n = 0;
        while (n < 10)
        {
            if (n == 3)
            {
                n++;
                continue;
            }
            if (n == 8)
                break;
            Console.WriteLine("Number is {0}", n);
            n++;
        }
    }
}
```

This code generates the following output:

```
0
1
2
4
5
6
7
```

do

A `do` loop functions just like a `while` loop, except the condition is evaluated at the end of the loop rather than the beginning of the loop:

```
using System;
class Test
{
    public static void Main()
    {
        int n = 0;
        do
```

```
        {
            Console.WriteLine("Number is {0}", n);
            n++;
        } while (n < 10);
    }
}
```

Like the `while` loop, the `break` and `continue` statements can control the flow of execution in the loop.

for

A `for` loop iterates over several values. You can declare the loop variable as part of the `for` statement:

```
using System;
class Test
{
    public static void Main()
    {
        for (int n = 0; n < 10; n++)
            Console.WriteLine("Number is {0}", n);
    }
}
```

The scope of the loop variable in a `for` loop is the scope of the statement or statement block that follows the `for`. It can't be accessed outside the loop structure:

```
// error
using System;
class Test
{
    public static void Main()
    {
        for (int n = 0; n < 10; n++)
        {
            if (n == 8)
                break;
            Console.WriteLine("Number is {0}", n);
        }
            // error; n is out of scope
        Console.WriteLine("Last Number is {0}", n);
    }
}
```

As with the `while` loop, the `break` and `continue` statements can control the flow of execution in the loop.

foreach

This is a common looping idiom:

```
using System;
using System.Collections;
class MyObject
{
}
class Test
{
    public static void Process(ArrayList arr)
    {
        for (int nIndex = 0; nIndex < arr.Count; nIndex++)
        {
                // cast is required by ArrayList stores
                // object references
            MyObject current = (MyObject) arr[nIndex];
            Console.WriteLine("Item: {0}", current);
        }
    }
}
```

This works fine, but it requires the programmer to ensure that the array in the `for` statement matches the array that's used in the indexing operation. If they don't match, it can sometimes be difficult to track down the bug. It also requires declaring a separate index variable, which could accidentally be used elsewhere.

It's also a lot of typing.

Some languages[3] provide a different construct for dealing with this problem, and C# also provides such a construct. You can rewrite the preceding example as follows:

```
using System;
using System.Collections;
class MyObject
{
}
class Test
{
    public static void Process(ArrayList arr)
    {
        foreach (MyObject current in arr)
        {
            Console.WriteLine("Item: {0}", current);
        }
    }
}
```

3. Depending on your language background, this might be Perl or Visual Basic.

This is a lot simpler, and it doesn't have the same opportunities for mistakes. The type returned by the index operation on `arr` is explicitly converted to the type declared in the `foreach`. This is nice, because collection types such as `ArrayList` can store values of type `object` only.

`foreach` also works for objects other than arrays. In fact, it works for any object that implements the proper interfaces. It can, for example, iterate over the keys of a hash table:

```
using System;
using System.Collections;
class Test
{
    public static void Main()
    {
        Hashtable    hash = new Hashtable();
        hash.Add("Fred", "Flintstone");
        hash.Add("Barney", "Rubble");
        hash.Add("Mr.", "Slate");
        hash.Add("Wilma", "Flintstone");
        hash.Add("Betty", "Rubble");

        foreach (string firstName in hash.Keys)
        {
            Console.WriteLine("{0} {1}", firstName, hash[firstName]);
        }
    }
}
```

User-defined objects can be implemented so that they can be iterated over using `foreach`; see Chapter 20 for more information. Of particular interest in Chapter 20 is the new C# 2.0 feature called iterators that makes implementing enumeration much easier. Iterators use the new context-dependant keyword `yield` for implementation.

The one thing you can't do in a `foreach` loop is change the contents of the container. In other words, in the previous example, the `firstName` variable can't be modified. If the container supports indexing, the contents could be changed through that route, though many containers that enable use by `foreach` don't provide indexing. Another thing to watch is to make sure the container isn't modified during the `foreach`; the behavior in such situations is undefined.[4]

As with other looping constructs, `break` and `continue` can be used with the `foreach` statement.

Jump Statements

Jump statements are used to do just that—jump from one statement to another.

break

The `break` statement breaks out of the current iteration or `switch` statement and continues execution after that statement.

4. A container could throw an exception in this case; however, that may be an expensive way to detect the condition.

continue

The `continue` statement skips all the later lines in the current iteration statement and then continues executing the iteration statement.

goto

The `goto` statement can jump directly to a label. Because using `goto` statements is widely considered to be harmful,[5] C# prohibits some of their worst abuses. A `goto` can't be used to jump into a statement block, for example. The only place where their use is recommended is in `switch` statements or to transfer control to outside a nested loop (though they can be used elsewhere).

return

The `return` statement returns to the calling function and optionally returns a value as well.

Other Statements

The following statements are covered in other chapters.

lock

The `lock` statement provides exclusive access to a thread; see Chapter 31.

using

The `using` statement is used in two ways. The first is to specify namespaces, which is covered in Chapter 3. The second use is to ensure that `Dispose()` is called at the end of a block, which is covered in detail in Chapter 8.

try-catch-finally

The `try`, `catch`, and `finally` statements control exception handling; see Chapter 4.

checked/unchecked

The `checked` and `unchecked` statements control whether exceptions are thrown if conversions or expressions overflow; see Chapter 14.

5. See "Go To Statement Considered Harmful," by Edsger W. Dijkstra, at `http://www.acm.org/classics/oct95/`.

CHAPTER 13

Variable Scoping and Definite Assignment

In C#, you can give local variables only those names that allow them to be uniquely identified in a given scope. If a name has more than one meaning in a scope and you have no way to disambiguate the name, the innermost declaration of the name is an error and you must change it. Consider the following:

```
using System;
class MyObject
{
    public MyObject(int x, int y)
    {
        this.x = x;
        this.y = y;
    }
    int x;
    int y;
}
```

In the constructor, x refers to the parameter named x because parameters take precedence over member variables. To access the instance variable named x, it must be prefixed with `this.`, which indicates it must be an instance variable.

The preceding construct is preferred over renaming the constructor parameters or member variables to avoid the name conflict.[1]

In the following situation, you have no way to name both variables, and the inner declaration is therefore an error:

```
// error
using System;
class MyObject
{
```

1. This is really a style thing, but in general it's considered to be cleaner.

```
    public void Process()
    {
        int    x = 12;
        for (int y = 1; y < 10; y++)
        {
                // no way to name outer x here.
            int x = 14;
            Console.WriteLine("x = {0}", x);
        }
    }
}
```

Because the inner declaration of `x` would hide the outer declaration of `x`, it isn't allowed.

C# has this restriction to improve code readability and maintainability. If this restriction wasn't in place, it might be difficult to determine which version of the variable was being used—or even that there *are* multiple versions—inside a nested scope.

Definite Assignment

Definite assignment rules prevent the value of an unassigned variable from being observed. Consider the following:

```
// error
using System;
class Test
{
    public static void Main()
    {
        int n;
        Console.WriteLine("Value of n is {0}", n);
    }
}
```

When this is compiled, the compiler will report an error because the value of `n` is used before it has been initialized.

Similarly, you can't perform operations with a class variable before it's initialized:

```
// error
using System;
class MyClass
{
    public MyClass(int value)
    {
        this.value = value;
    }
```

```
    public int Calculate()
    {
        return(value * 10);
    }
    public int    value;
}
class Test
{
    public static void Main()
    {
        MyClass mine;

        Console.WriteLine("{0}", mine.value);            // error
        Console.WriteLine("{0}", mine.Calculate());        // error
        mine = new MyClass(12);
        Console.WriteLine("{0}", mine.value);        // okay now…
    }
}
```

Structs work slightly differently when you consider definite assignment. The runtime will always make sure they're zeroed out, but the compiler will still check to make sure they're initialized to a value before they're used.

You initialize a struct either by calling a constructor or by setting all the members of an instance before it's used:

```
using System;
struct Complex
{
    public Complex(float real, float imaginary)
    {
        this.real = real;
        this.imaginary = imaginary;
    }
    public override string ToString()
    {
        return(String.Format("({0}, {1})", real, imaginary));
    }

    public float    real;
    public float    imaginary;
}
```

```
class Test
{
    public static void Main()
    {
        Complex    myNumber1;
        Complex    myNumber2;
        Complex    myNumber3;

        myNumber1 = new Complex();
        Console.WriteLine("Number 1: {0}", myNumber1);

        myNumber2 = new Complex(5.0F, 4.0F);
        Console.WriteLine("Number 2: {0}", myNumber2);

        myNumber3.real = 1.5F;
        myNumber3.imaginary = 15F;
        Console.WriteLine("Number 3: {0}", myNumber3);
    }
}
```

In the first section of this code, `myNumber1` is initialized by the call to `new`. Remember, structs don't have default constructors, so this call doesn't do anything; it merely has the side effect of marking the instance as initialized.

In the second section, `myNumber2` is initialized by a normal call to a constructor.

In the third section, `myNumber3` is initialized by assigning values to all members of the instance. Obviously, you can do this only if the members are public.

Definite Assignment and Arrays

Arrays work a bit differently for definite assignment. For arrays of both reference and value types (classes and structs), an element of an array *can* be accessed, even if it hasn't been initialized to a nonzero value.

For example, suppose you have an array of `Complex`:

```
using System;
struct Complex
{
    public Complex(float real, float imaginary)
    {
        this.real = real;
        this.imaginary = imaginary;
    }
    public override string ToString()
    {
        return(String.Format("({0}, {0})", real, imaginary));
    }
```

```
    public float    real;
    public float    imaginary;
}

class Test
{
    public static void Main()
    {
        Complex[]    arr = new Complex[10];
        Console.WriteLine("Element 5: {0}", arr[5]);        // legal
    }
}
```

Because of the operations that might be performed on an array—such as `Reverse()`—the compiler can't track definite assignment in all situations, and it could lead to spurious errors. It therefore doesn't try.

CHAPTER 14

Operators and Expressions

The C# expression syntax is based upon the C++ expression syntax.

Operator Precedence

When an expression contains multiple operators, the precedence of the operators controls the order in which the elements of the expression are evaluated. You can change the default precedence by grouping elements with parentheses, like so:

```
int    value = 1 + 2 * 3;        // 1 + (2 * 3) = 7
       value = (1 + 2) * 3;      // (1 + 2) * 3 = 9
```

In C#, all binary operators are left-associative, which means operations are performed from left to right, except for the assignment and conditional (`?:`) operators, which are performed from right to left.

Table 14-1 summarizes all the operators in precedence from highest to lowest.

Table 14-1. *Operator Precedence*

Category	Operators
Primary	`(x) x.y f(x) a[x] x++ x-- new typeof sizeof checked unchecked`
Unary	`+ - ! ~ ++x --x (T)x`
Multiplicative	`* / %`
Additive	`+ -`
Shift	`<< >>`
Relational	`< > <= >= is`
Equality	`== !=`
Logical AND	`&`
Logical XOR	`^`
Logical OR	`|`
Conditional AND	`&&`
Conditional OR	`||`

Table 14-1. *Operator Precedence (Continued)*

Category	Operators
Conditional	`?:`
Assignment	`= *= /= %= += -= <<= >>= &= ^= |=`

Built-in Operators

For numeric operations in C#, the `int`, `uint`, `long`, `ulong`, `float`, `double`, and `decimal` types typically have built-in operators. Because other types don't have built-in operators, you must first convert an expression to one of the types that has an operator before the operation is performed.

A good way to think about this is to consider that an operator (+ in this case)[1] has the following built-in overloads:

```
int operator +(int x, int y);
uint operator +(uint x, uint y);
long operator +(long x, long y);
ulong operator +(ulong x, ulong y);
float operator +(float x, float y);
double operator +(double x, double y);
```

Notice that these operations all take two parameters of the same type and return that type. For the compiler to perform an addition, it can use only one of these functions. This means smaller sizes (such as two `short` values) can't be added without them being converted to `int`, and such an operation will return `int`.

The result of this is that when operations are done with numeric types that don't have a built-in operator but can be implicitly converted to a type that does, the result of the operation will be "larger" than the two types that provided input into the operator. This requires the result to be explicitly cast back to the "smaller" type.[2]

```
// error
class Test
{
    public static void Main()
    {
        short    s1 = 15;
        short    s2 = 16;
        short ssum = (short) (s1 + s2);   // cast is required

        int i1 = 15;
        int i2 = 16;
        int isum = i1 + i2;               // no cast required
    }
}
```

1. There's also the overload for strings, but that's outside the scope of this example.
2. You may object to this, but you really wouldn't like the type system of C# if it didn't work this way.

User-Defined Operators

You can declare user-defined operators for classes or structs, and they function in the same manner in which the built-in operators function. In other words, you can define the + operator on a class or struct so that an expression such as `a + b` is valid. In the following sections, the operators that can be overloaded are marked with "over" in subscript. See Chapter 26 for more information.

Numeric Promotions

See Chapter 15 for information on the rules for numeric promotion.

Arithmetic Operators

The following sections summarize the arithmetic operations that can be performed in C#. The floating-point types have specific rules to follow;[3] for full details, see the CLR. If executed in a checked context, arithmetic expressions on nonfloating types may throw exceptions.

Unary Plus (+) $_{\text{over}}$

For unary plus, the result is simply the value of the operand.

Unary Minus (-) $_{\text{over}}$

Unary minus works only on types that have a valid negative representation, and it returns the value of the operand subtracted from zero.

Bitwise Complement (~) $_{\text{over}}$

The ~ operator returns the bitwise complement of a value.

Addition (+) $_{\text{over}}$

In C#, the + operator is used both for addition and for string concatenation.

Numeric Addition

The two operands are added together. If the expression is evaluated in a checked context and the sum is outside the range of the result type, an `OverflowException` is thrown. The following code demonstrates this:

3. They conform to IEEE 754 arithmetic.

```
using System;
class Test
{
    public static void Main()
    {
        byte val1 = 200;
        byte val2 = 201;
        byte sum = (byte) (val1 + val2);            // no exception
        checked
        {
            byte sum2 = (byte) (val1 + val2);      // exception
        }
    }
}
```

String Concatenation

You can perform string concatenation between two strings or between a string and an operand of type `object`.[4] If either operand is null, an empty string is substituted for that operand.

Operands that aren't of type `string` will be automatically converted to a string by calling the virtual `ToString()` method on the object.

Subtraction (-) over

The second operand is subtracted from the first operand. If the expression is evaluated in a checked context and the difference is outside the range of the result type, an `OverflowException` is thrown.

Multiplication (*) over

The two operands are multiplied together. If the expression is evaluated in a checked context and the result is outside the range of the result type, an `OverflowException` is thrown.

Division (/) over

The first operand is divided by the second operand. If the second operand is zero, a `DivideByZero` exception is thrown.

Remainder (%) over

The result `x % y` is computed as `x - (x / y) * y` using integer operations. If `y` is zero, a `DivideByZero` exception is thrown.

4. Since any type can convert to `object`, this means any type.

Shift (<< and >>) over

For left shifts, the high-order bits are discarded and the low-order empty bit positions are set to zero.

For right shifts with `uint` or `ulong`, the low-order bits are discarded and the high-order empty bit positions are set to zero.

For right shifts with `int` or `long`, the low-order bits are discarded, and the high-order empty bit positions are set to `0` if `x` is non-negative and to `1` if `x` is negative.

Increment and Decrement (++ and --) over

The increment operator increases the value of a variable by 1, and the decrement operator decreases the value of the variable by 1.[5]

Increment and decrement can be used either as a prefix operator, where the variable is modified before it's read, or as a postfix operator, where the value is returned before it's modified.

For example:

```
int    k = 5;
int    value = k++;    // value is 5
       value = --k;    // value is still 5
       value = ++k;    // value is 6
```

Note that increment and decrement are exceptions to the rule about smaller types requiring casts to function. A cast is required when adding two shorts and assigning them to another short:

```
short s = (short) a + b;
```

Such a cast isn't required for an increment of a short:[6]

```
s++;
```

Relational and Logical Operators

Relational operators compare two values, and logical operators perform bitwise operations on values.

Logical Negation (!) over

The `!` operator returns the negation of a Boolean value.

5. In unsafe mode, pointers increment and decrement by the size of the pointed-to object.
6. In other words, there are predefined increment and decrement functions for the types smaller than `int` and `uint`.

Relational Operators over

C# defines the relational operations shown in Table 14-2.

Table 14-2. *Relational Operations*

Operation	Description
`a == b`	Returns true if a is equal to b
`a != b`	Returns true if a is not equal to b
`a < b`	Returns true if a is less than b
`a <= b`	Returns true if a is less than or equal to b
`a > b`	Returns true if a is greater than b
`a >= b`	Returns true if a is greater than or equal to b

These operators return a result of type `bool`.

When performing a comparison between two reference-type objects, the compiler will first look for relational operators defined on the objects (or base classes of the objects). If it finds no applicable operator, and the relational is `==` or `!=`, the appropriate relational operator will be called from the `object` class. This operator compares whether the two operands reference the same instance, not whether they have the same value.

For value types, the process is the same if `==` and `!=` are overloaded. If they aren't overloaded, there's no default implementation for value types, and an error is generated.

The overloaded versions of `==` and `!=` are closely related to the `Object.Equals()` member. See Chapter 29 for more information.

For the `string` type, the relational operators are overloaded, so `==` and `!=` compare the values of the strings, not the references.

Logical Operators over

C# defines the logical operators shown in Table 14-3.

Table 14-3. *Logical Operators*

Operator	Description
&	Bitwise AND of the two operands
|	Bitwise OR of the two operands
^	Bitwise exclusive OR (XOR) of the two operands
&&	Logical AND of the two operands
||	Logical OR of the two operands

The operators &, |, and ^ are usually used on integer data types, though they can also be applied to the bool type.

The operators && and || differ from the single-character versions in that they perform short-circuit evaluation. In the following expression, b is evaluated only if a is true:

```
a && b
```

In the following expression, b is evaluated only if a is false:

```
a || b
```

Conditional Operator (?:)

Sometimes called the *ternary* or *question* operator, the conditional operator selects from two expressions based on a Boolean expression:

```
int    value = (x < 10) ? 15 : 5;
```

In this example, the control expression (x < 10) is evaluated. If it's true, the value of the operator is the first expression following the question mark, which is 15 in this case. If the control expression is false, the value of the operator is the expression following the colon, which is 5 in this case.

Assignment Operators

Assignment operators assign a value to a variable. They come in two forms: the simple assignment and the compound assignment.

Simple Assignment

You can perform simple assignment in C# using the single equals (=) sign. For the assignment to succeed, the right side of the assignment must be a type that can be implicitly converted to the type of the variable on the left side of the assignment.

Compound Assignment

Compound assignment operators perform some operation in addition to simple assignment. The following are the compound operators:

```
+=-=*=/=%=&= |=^=<<=>>=
```

The compound operator x <op>= y is evaluated exactly as if it were written as x = x <op> y with two exceptions:

- x is evaluated only once, and that evaluation is used for both the operation and the assignment.
- If x contains a function call or array reference, it's performed only once.

Under normal conversion rules, if x and y are both short integers, evaluating `x = x + 3;` would produce a compile-time error, because addition is performed on `int` values and the `int` result isn't implicitly converted to a `short`. In this case, however, because `short` can be implicitly converted to `int` and it's possible to write the following, the operation is permitted:

```
x = 3;
```

Type Operators

Rather than dealing with the values of an object, the type operators deal with the type of an object.

typeof

The `typeof` operator returns the type of the object, which is an instance of the `System.Type` class. `Typeof` is useful to avoid having to create an instance of an object just to obtain the `type` object. If an instance already exists, a `type` object can be obtained by calling the `GetType()` function on the instance.

Once you've obtained the `type` object for a type, you can query it using reflection to obtain information about the type. See Chapter 38 for more information.

is

The `is` operator determines whether an object reference can be converted to a specific type or interface. The most common use of this operator is to determine whether an object supports a specific interface:

```
using System;
interface IAnnoy
{
    void PokeSister(string name);
}
class Brother: IAnnoy
{
    public void PokeSister(string name)
    {
        Console.WriteLine("Poking {0}", name);
    }
}
class BabyBrother
{
}
class Test
{
    public static void AnnoyHer(string sister, params object[] annoyers)
    {
        foreach (object o in annoyers)
        {
```

```
            if (o is IAnnoy)
            {
                IAnnoy annoyer = (IAnnoy) o;
                annoyer.PokeSister(sister);
            }
        }
    }
    public static void Main()
    {
        Test.AnnoyHer("Jane", new Brother(), new BabyBrother());
    }
}
```

This code produces the following output:

```
Poking: Jane
```

In this example, the `Brother` class implements the `IAnnoy` interface, and the `BabyBrother` class doesn't. The `AnnoyHer()` function walks through all the objects that are passed to it, checks to see if an object supports `IAnnoy`, and then calls the `PokeSister()` function if the object supports the interface.

as

The `as` operator is similar to the `is` operator, but instead of just determining whether an object is a specific type or interface, it also performs the explicit conversion to that type. If the object can't be converted to that type, the operator returns null. Using `as` is more efficient than the `is` operator, since the `as` operator needs to check the type of the object only once, while the example using `is` checks the type when the operator is used and again when the conversion is performed.

In the previous example, you could replace these lines:

```
if (o is IAnnoy)
{
    IAnnoy annoyer = (IAnnoy) o;
    annoyer.PokeSister(sister);
}
```

with the following ones:

```
IAnnoy annoyer = o as IAnnoy;
if (annoyer != null)
    annoyer.PokeSister(sister);
```

Note that you can't use the `as` operator with boxed value types. The following doesn't work, because there's no way to get a null value as a value type:

```
int value = o as int;
```

checked and unchecked Expressions

When dealing with expressions, it's often difficult to strike the right balance between the performance of expression evaluation and the detection of overflow in expressions or conversions. Some languages choose performance and can't detect overflow, and other languages put up with reduced performance and always detect overflow.

In C#, the programmer is able to choose the appropriate behavior for a specific situation using the `checked` and `unchecked` keywords.

Code that depends upon the detection of overflow can be wrapped in a `checked` block:

```
using System;

class Test
{
    public static void Main()
    {
        checked
        {
            byte a = 55;
            byte b = 210;
            byte c = (byte) (a + b);
        }
    }
}
```

When this code is compiled and executed, it will generate an `OverflowException`.

Similarly, if the code depends on the truncation behavior, the code can be wrapped in an `unchecked` block:

```
using System;

class Test
{
    public static void Main()
    {
        unchecked
        {
            byte a = 55;
            byte b = 210;
            byte c = (byte) (a + b);
        }
    }
}
```

For the remainder of the code, you can control the behavior with the `/checked+` compiler switch. Usually, `/checked+` is turned on for debug builds to catch possible problems and then turned off in retail builds to improve performance.

CHAPTER 15

Conversions

In C#, conversions are divided into implicit and explicit conversions. Implicit conversions are those that will always succeed; the conversion can always be performed without data loss.[1] For numeric types, this means the destination type can fully represent the range of the source type. For example, a `short` can be converted implicitly to an `int`, because the `short` range is a subset of the `int` range.

Numeric Types

For the numeric types, there are widening implicit conversions for all the signed and unsigned numeric types. Figure 15-1 shows the conversion hierarchy. If a path of arrows can be followed from a source type to a destination type, there's an implicit conversion from the source to the destination. For example, there are implicit conversions from `sbyte` to `short`, from `byte` to `decimal`, and from `ushort` to `long`.

Note that the path taken from a source type to a destination type in the figure doesn't represent how the conversion happens; it merely indicates it can be done. In other words, the conversion from `byte` to `long` happens in a single operation, not by converting through `ushort` and `uint`:

```
class Test
{
    public static void Main()
    {
            // all implicit
        sbyte v = 55;
        short v2 = v;
        int v3 = v2;
        long v4 = v3;

            // explicit to "smaller" types
        v3 = (int) v4;
        v2 = (short) v3;
        v = (sbyte) v2;
    }
}
```

1. Conversions from `int`, `uint`, or `long` to `float`, and conversions from `long` to `double` may result in a loss of precision but won't result in a loss of magnitude.

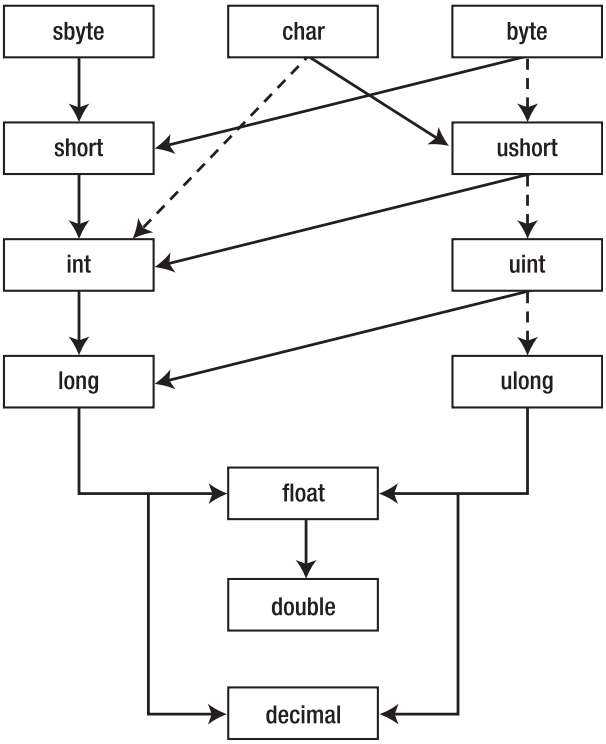

Figure 15-1. *C# conversion hierarchy*

Conversions and Member Lookup

When considering overloaded members, the compiler may have to choose between several functions. Consider the following:

```
using System;
class Conv
{
    public static void Process(sbyte value)
    {
        Console.WriteLine("sbyte {0}", value);
    }
    public static void Process(short value)
    {
        Console.WriteLine("short {0}", value);
    }
    public static void Process(int value)
    {
        Console.WriteLine("int {0}", value);
    }
}
```

```
class Test
{
    public static void Main()
    {
        int    value1 = 2;
        sbyte    value2 = 1;
        Conv.Process(value1);
        Conv.Process(value2);
    }
}
```

The preceding code produces the following output:

```
int 2
sbyte 1
```

In the first call to `Process()`, the compiler could match the `int` parameter to only one of the functions—the one that took an `int` parameter.

In the second call, however, the compiler had three versions to choose from, taking `sbyte`, `short`, or `int`. To select one version, it first tries to match the type exactly. In this case, it can match `sbyte`, so that's the version that gets called. If the `sbyte` version didn't appear there, it'd select the `short` version, because a `short` can be converted implicitly to an `int`. In other words, `short` is "closer to" `sbyte` in the conversion hierarchy and is therefore preferred.

The preceding rule handles many cases, but it doesn't handle the following one:

```
using System;
class Conv
{
    public static void Process(short value)
    {
        Console.WriteLine("short {0}", value);
    }
    public static void Process(ushort value)
    {
        Console.WriteLine("ushort {0}", value);
    }
}
class Test
{
    public static void Main()
    {
        byte    value = 3;
        Conv.Process(value);
    }
}
```

Here, the earlier rule doesn't allow the compiler to choose one function over the other, because there are no implicit conversions in either direction between `ushort` and `short`.

In this case, there's another rule that kicks in, which says that if there's a single-arrow implicit conversion to a signed type, it will be preferred over all conversions to unsigned types. Figure 15-1 represents this graphically with dotted arrows; the compiler will choose a single solid arrow over any number of dotted arrows.

Explicit Numeric Conversions

Explicit conversions—those using the cast syntax—are the conversions that operate in the opposite direction from the implicit conversions. Converting from `short` to `long` is implicit; therefore, converting from `long` to `short` is an explicit conversion.

Viewed another way, an explicit numeric conversion may result in a value that's different from the original:

```
using System;
class Test
{
    public static void Main()
    {
        uint value1 = 312;
        byte value2 = (byte) value1;
        Console.WriteLine("Value2: {0}", value2);
    }
}
```

The preceding code results in the following output:

```
56
```

In the conversion to `byte`, the least significant (lowest valued) part of the `uint` is put into the `byte` value. In many cases, the programmer either knows that the conversion will succeed or depends on this behavior.

Checked Conversions

In other cases, it may be useful to check whether the conversion succeeded. You can do this by executing the conversion in a `checked` context:

```
using System;
class Test
{
    public static void Main()
    {
        checked
        {
```

```
            uint value1 = 312;
            byte value2 = (byte) value1;
            Console.WriteLine("Value: {0}", value2);
        }
    }
}
```

When an explicit numeric conversion takes place in a `checked` context, if the source value won't fit in the destination data type, an exception will be thrown.

The `checked` statement creates a block in which conversions are checked for success. Whether a conversion is checked is determined at compile time, and the checked state doesn't apply to code in functions called from within the `checked` block.

Checking conversions for success does have a small performance penalty and therefore may not be appropriate for released software. It can, however, be useful to check all explicit numeric conversions when developing software. The C# compiler provides a `/checked` compiler option that will generate checked conversions for all explicit numeric conversions. This option can be used while developing software and then can be turned off to improve performance for released software.

If the programmer is depending upon the unchecked behavior, turning on `/checked` could cause problems. In this case, the `unchecked` statement can indicate that none of the conversions in a block should ever be checked for conversions.

It's sometimes useful to be able to specify the checked state for a single statement; in this case, you can specify the `checked` or `unchecked` operator at the beginning of an expression:

```
using System;
class Test
{
    public static void Main()
    {
        uint value1 = 312;
        byte value2;

        value2 = unchecked((byte) value1);    // never checked
        value2 = (byte) value1;               // checked if /checked
        value2 = checked((byte) value1);      // always checked
    }
}
```

In this example, the first conversion will never be checked, the second conversion will be checked if the `/checked` statement is present, and the third conversion will always be checked.

Conversions of Classes (Reference Types)

Conversions involving classes are similar to those involving numeric values, except that object conversions deal with casts up and down the object inheritance hierarchy instead of conversions up and down the numeric type hierarchy.

C# also allows conversion between unrelated classes (or structs) to be overloaded. We discuss this later in this chapter.

As with numeric conversions, implicit conversions are those that will always succeed, and explicit conversions are those that may fail.

To the Base Class of an Object

A reference to an object can be converted implicitly to a reference to the base class of an object. Note that this *doesn't* convert the object to the type of the base class; only the reference is to the base class type. The following example illustrates this:

```
using System;
public class Base
{
    public virtual void WhoAmI()
    {
        Console.WriteLine("Base");
    }
}
public class Derived: Base
{
    public override void WhoAmI()
    {
        Console.WriteLine("Derived");
    }
}
public class Test
{
    public static void Main()
    {
        Derived d = new Derived();
        Base b = d;

        b.WhoAmI();
        Derived d2 = (Derived) b;

        object o = d;
        Derived d3 = (Derived) o;
    }
}
```

This code produces the following output:

```
Derived
```

Initially, a new instance of `Derived` is created, and the variable `d` contains a reference to that object. The reference `d` is then converted to a reference to the base type `Base`. The object referenced by both variables, however, is still a `Derived`; when the virtual function `WhoAmI()` is called, the version from `Derived` is called.[2] It's also possible to convert the `Base` reference `b` back to a reference of type `Derived` or to convert the `Derived` reference to an `object` reference and back.

Converting to the base type is an implicit conversion because, as discussed in Chapter 1, a derived class *is* always an example of the base class. In other words, `Derived` "is-a" `Base`.

Explicit conversions are possible between classes when a "could-be" relationship exists. Because `Derived` is derived from `Base`, any reference to `Base` could really be a `Base` reference to a `Derived` object, and therefore the conversion can be attempted. At runtime, the actual type of the object referenced by the `Base` reference (`b` in the previous example) will be checked to see if it's really a reference to `Derived`. If it isn't, an exception will be thrown on the conversion.

Because `object` is the ultimate base type, any reference to a class can be implicitly converted to a reference to `object`, and a reference to `object` may be explicitly converted to a reference to any class type.

Figure 15-2 illustrates the previous example.

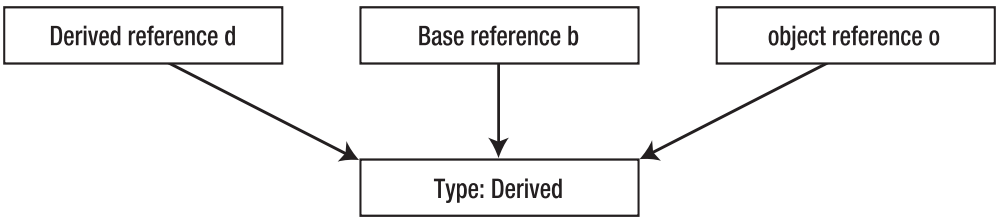

Figure 15-2. *Different references to the same instance*

To an Interface the Object Implements

Interface implementation is somewhat like class inheritance. If a class implements an interface, an implicit conversion can be used to convert from a reference to an instance of the class to the interface. This conversion is implicit because it's known at compile time that it works.

Once again, the conversion to an interface doesn't change the underlying type of an object. A reference to an interface can therefore be converted explicitly back to a reference to an object that implements the interface, since the interface reference "could-be" referencing an instance of the specified object.

In practice, converting from the interface to an object is an operation that's rarely, if ever, used.

To an Interface the Object Might Implement

The implicit conversion from an object reference to an interface reference discussed in the previous section isn't common. An interface is especially useful in situations where it isn't known whether an object implements an interface. The following example implements a debug trace routine that uses an interface if it's available:

2. Similarly, `Type.GetType`, `is`, and `as` would also show it to be a derived instance.

```
using System;
interface IDebugDump
{
    string DumpObject();
}
class Simple
{
    public Simple(int value)
    {
        this.value = value;
    }
    public override string ToString()
    {
        return(value.ToString());
    }
    int value;
}
class Complicated: IDebugDump
{
    public Complicated(string name)
    {
        this.name = name;
    }
    public override string ToString()
    {
        return(name);
    }
    string IDebugDump.DumpObject()
    {
        return(String.Format(
            "{0}\nLatency: {1}\nRequests: {2}\nFailures: {3}\n",
            new object[] {name,    latency, requestCount, failedCount} ));
    }
    string name;
    int latency = 0;
    int requestCount = 0;
    int failedCount = 0;
}
class Test
{
    public static void DoConsoleDump(params object[] arr)
    {
        foreach (object o in arr)
        {
            IDebugDump dumper = o as IDebugDump;
            if (dumper != null)
                Console.WriteLine("{0}", dumper.DumpObject());
```

```
            else
                Console.WriteLine("{0}", o);
        }
    }
    public static void Main()
    {
        Simple s = new Simple(13);
        Complicated c = new Complicated("Tracking Test");
        DoConsoleDump(s, c);
    }
}
```

This produces the following output:

```
13
Tracking Test
Latency: 0
Requests: 0
Failures: 0
```

This example has dumping functions that can list objects and their internal state. Some objects have a complicated internal state and need to pass back some rich information, while others can get by with the information returned by their `ToString()` functions.

This is nicely expressed by the `IDebugDump` interface, which is used to generate the output if an implementation of the interface is present.

This example uses the `as` operator, which will return the interface if the object implements it or null if it doesn't.

From One Interface Type to Another

A reference to an interface can be converted implicitly to a reference to an interface that it's based upon. It can be converted explicitly to a reference to any interface that it isn't based upon. This is successful only if the interface reference is a reference to an object that implements the other interface as well.

Conversions of Structs (Value Types)

The only built-in conversion dealing with structs is an implicit conversion from a struct to an interface that it implements. The instance of the struct will be boxed to a reference and then converted to the appropriate interface reference. There are no implicit or explicit conversions from an interface to a struct.

CHAPTER 16

Arrays

Arrays in C# are reference objects; they're allocated out of heap space rather than on the stack. The elements of an array are stored as dictated by the element type; if the element type is a reference type (such as `string`), the array will store references to strings. If the element is a value type (such as a numeric type or a struct type), the elements are stored directly within the array. In other words, an array of a value type doesn't contain boxed instances.

You can declare arrays using the following syntax:

```
<type>[] identifier;
```

The initial value of an array is `null`. You can create an array object using `new`:

```
int[] store = new int[50];
string[] names = new string[50];
```

When an array is created, it initially contains the default values for the types that are in the array. For the `store` array, each element is an `int` with the value `0`. For the `names` array, each element is a `string` reference with the value `null`.

Array Initialization

You can initialize arrays at the same time as you create them. During initialization, you can omit `new int[x]`, and the compiler will determine the size of the array to allocate from the number of items in the initialization list:

```
int[] store = {0, 1, 2, 3, 10, 12};
```

The preceding line is equivalent to this:

```
int[] store = new int[6] {0, 1, 2, 3, 10, 12};
```

Multidimensional and Jagged Arrays

To index elements in more than one dimension, you can use either a multidimensional array or a jagged array.

Multidimensional Arrays

Multidimensional arrays have more than one dimension:

```
int[,] matrix = new int[4, 2];
matrix[0, 0] = 5;
matrix[3, 1] = 10;
```

The `matrix` array has a first dimension of `4` and a second dimension of `2`. This array could be initialized using the following statement:

```
int[,] matrix = { {1, 1}, {2, 2}, {3, 5}, {4, 5}};
```

The `matrix` array has a first dimension of `5` and a second dimension of `2`.

Multidimensional arrays are sometimes called *rectangular arrays* because the elements can be written in a rectangular table (for dimensions less than `2`). When the matrix array is allocated, a single chunk is obtained from the heap to store the entire array (see Figure 16-1).

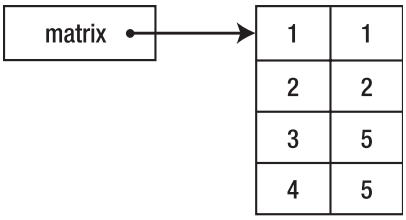

Figure 16-1. *Storage in a multidimensional array*

The following is an example of using a multidimensional array:

```
using System;
class Test
{
    public static void Main()
    {
        int[,] matrix = { {1, 1}, {2, 2}, {3, 5}, {4, 5}, {134, 44} };

        for (int i = 0; i < matrix.GetLength(0); i++)
        {
            for (int j = 0; j < matrix.GetLength(1); j++)
            {
                Console.WriteLine("matrix[{0}, {1}] = {2}", i, j, matrix[i, j]);
            }
        }
    }
}
```

The `GetLength()` member of an array will return the length of that dimension. This example produces the following output:

```
matrix[0, 0] = 1
matrix[0, 1] = 1
matrix[1, 0] = 2
matrix[1, 1] = 2
matrix[2, 0] = 3
matrix[2, 1] = 5
matrix[3, 0] = 4
matrix[3, 1] = 5
matrix[4, 0] = 134
matrix[4, 1] = 44
```

Jagged Arrays

A *jagged array* is merely an array of arrays and is called *jagged* because it doesn't have to be rectangular. For example:

```
int[][] matrix = new int[3][];
matrix[0] = new int[5];
matrix[1] = new int[4];
matrix[2] = new int[2];
matrix[0][3] = 4;
matrix[1][1] = 8;
matrix[2][0] = 5;
```

The `matrix` array here has only a single dimension of three elements. Its elements are integer arrays. The first element is an array of five integers, the second is an array of four integers, and the third is an array of two integers (see Figure 16-2). The `matrix` variable is a reference to an array of three references to arrays of integers. Four heap allocations were required for this array.

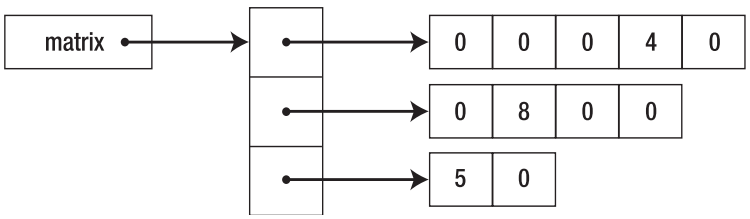

Figure 16-2. *Storage in a jagged array*

Using the initialization syntax for arrays, you could write the following:

```
using System;
class Test
{
    public static void Main()
    {
        int[][] matrix = {new int[5], new int[4], new int[2] };
        matrix[0][3] = 4;
        matrix[1][1] = 8;
        matrix[2][0] = 5;

        for (int i = 0; i < matrix.Length; i++)
        {
            for (int j = 0; j < matrix[i].Length; j++)
            {
                Console.WriteLine("matrix[{0}, {1}] = {2}", i, j, matrix[i][j]);
            }
        }
    }
}
```

Note that the traversal code is different from the multidimensional case. Because `matrix` is an array of arrays, you need to use a nested single-dimensional traverse.

Arrays of Reference Types

Arrays of reference types can be somewhat confusing because the elements of the array are initialized to null rather than to the element type. For example:

```
class Employee
{
    public void LoadFromDatabase(int employeeID)
    {
        // load code here
    }
}
class Test
{
    public static void Main()
    {
        Employee[] emps = new Employee[3];
        emps[0].LoadFromDatabase(15);
        emps[1].LoadFromDatabase(35);
        emps[2].LoadFromDatabase(255);
    }
}
```

When LoadFromDatabase() is called, a null exception will be generated because the elements referenced have never been set and are therefore still null.

You can rewrite the class as follows:

```
class Employee
{
    public static Employee LoadFromDatabase(int employeeID)
    {
        Employee emp = new Employee();
        // load code here
        return(emp);
    }
}
class Test
{
    public static void Main()
    {
        Employee[] emps = new Employee[3];
        emps[0] = Employee.LoadFromDatabase(15);
        emps[1] = Employee.LoadFromDatabase(35);
        emps[2] = Employee.LoadFromDatabase(255);
    }
}
```

This allows you to create an instance, load it, and then save it into the array.

The reason that arrays aren't initialized is for performance. If the compiler did do the initialization, it'd need to do the same initialization for each element, and if that wasn't the right initialization, all those allocations would be wasted.

Array Conversions

Conversions are allowed between arrays based on the number of dimensions and the conversions available between the element types.

An implicit conversion is permitted from array S to array T if the arrays have the same number of dimensions, if the elements of S have an implicit reference conversion to the element type of T, and if both S and T are reference types. In other words, if an array of class references exists, it can be converted to an array of a base type of the class.

Explicit conversions have the same requirements, except that the elements of S must be explicitly convertible to the element type of T:

```
using System;
class Test
{
    public static void PrintArray(object[] arr)
    {
        foreach (object obj in arr)
            Console.WriteLine("Word: {0}", obj);
    }
```

```
    public static void Main()
    {
        string s = "I will not buy this record, it is scratched.";
        char[] separators = {' '};
        string[] words = s.Split(separators);
        PrintArray(words);
    }
}
```

In this example, the `string` array of words can be passed as an `object` array, because each `string` element can be converted to `object` through a reference conversion. This isn't possible, for example, if there's a user-defined implicit conversion.

The System.Array Type

Because arrays in C# are based on the .NET runtime `System.Array` type, they can perform several operations that aren't traditionally supported by array types.

Sorting and Searching

The ability to sort and search is built into the `System.Array` type. The `Sort()` function will sort the items of an array, and the `IndexOf()`, `LastIndexOf()`, and `BinarySearch()` functions are used to search for items in the array. For more information, see Chapter 30.

Reverse

Calling `Reverse()` simply reverses all the elements of the array:

```
using System;
class Test
{
    public static void Main()
    {
        int[] arr = {5, 6, 7};
        Array.Reverse(arr);
        foreach (int value in arr)
        {
            Console.WriteLine("Value: {0}", value);
        }
    }
}
```

This produces the following output:

```
7
6
5
```

CHAPTER 17

Generics

Without a doubt, generics are the biggest version 2.0 addition to the C# language. One of the biggest disappointments of the C# designers was their inability to add generic types (also sometimes known as *parameterized types*) to the first version of the language. The motivation for generics is simple—suppose a class wants to store a member variable, or a method wants to take a parameter, but the class of the input object should be specified by the client of the class or method, not by the author of the original code. Without generics, the only way to accomplish this is by providing the loosest possible typing, which means using `object` as the class for the member variable or method parameter. This approach has two problems: the lack of type-safety and the performance problems caused by boxing. Using generics solves these problems.

An Overview of Generics

For programmers coming to C# version 1 from a C++ background, the first feature request they often came up with was for generics to be included in the C# language. The prototypical use for generics is collection class libraries, but this powerful language feature can also be used in many other ways to build useful classes that would be difficult or impractical without generics.

C# generics differ from C++ templates in a number of ways. The key difference is that C# generics aren't a language-only feature and have explicit runtime support. This means other languages that support generics, such as Visual Basic .NET and C++/CLI, can consume a generic class written in C#. Generics don't support some of the more advanced features of C++ templates, such as partial template specification. In addition, C++ templates are a compile-time feature, while .NET generics are "expanded" at runtime.

Consider the following code sample that implements a growable array, written without using generics:

```
public class GrowableArray
{
 private object[] collection = new object[16];
 private int count = 0;

 public void AddElement(object element)
 {
```

```
  if (count >= collection.Length)
  {
   object[] temp = new object[collection.Length * 2];
   Array.Copy(collection, temp, collection.Length);
   collection = temp;
  }
  collection[count] = element;
  ++count;
 }

 public object GetElement(int elementNumber)
 {
  if (elementNumber >= count)
  {
   throw new IndexOutOfRangeException();
  }
  return collection[elementNumber];
 }
}
```

Using an internal `object` array allows any type to be used within the collection; however, as mentioned in Chapter 9, storing a value type using an `object` reference causes a boxing operation to occur. The return type of the `GetElement` method demonstrates the other problem with `object`-based collections—a cast will generally be required to convert the `object` reference to the specific type stored in the collection. The following code highlights these issues:

```
GrowableArray ga = new GrowableArray();
int num = 10;
ga.AddElement(num); //boxing operation here

//unboxing operation and no compile-time type-safety
int newNum = (int)ga.GetElement(0);

//runtime error here
string str = (string)ga.GetElement(0);
```

Generics solve the problems shown in the previous code sample. Instead of using an `object` reference to specify a "generic" parameter, generics allow the type of a parameter to be left unspecified until the generic code is used. You implement this by specifying a special parameter known as a *type parameter* inside angle brackets (< and >); type parameters signify that it's up to the client code to provide the actual type that will be substituted for the type parameter. If you rewrite the earlier `GrowableArray` sample using generics, you simply have to replace object references with the generic placeholder `T`:

```
public class GrowableArray<T>
{
 private T[] collection = new T[16];
 private int count = 0;

 public void AddElement(T element)
 {
  if (count >= collection.Length)
  {
   T[] temp = new T[collection.Length * 2];
   Array.Copy(collection, temp, collection.Length);
   collection = temp;
  }
  collection[count] = element;
  ++count;
 }

 public T GetElement(int elementNumber)
 {
  if (elementNumber >= count)
  {
   throw new IndexOutOfRangeException();
  }
  return collection[elementNumber];
 }
}
```

In this `GrowableArray` collection, a type parameter called `T` has been specified at the class level, and all the code that deals with the internal array has been converted from referencing `object` to referencing `T`. The author of the collection doesn't know the exact type of `T`, and code that uses the collection can specify any type that it wants to store in the collection. Code that uses the generic `GrowableArray` now looks like this:

```
GrowableArray<int> ga = new GrowableArray<int>();
int num = 10;
ga.AddElement(num); //no boxing operation here
int newNum = ga.GetElement(0); //type-safety and no unboxing operation
// string str = (string)ga.GetElement(0); //would not compile
```

`GrowableArray<int>` is known as a *constructed type*, and the type that the consumer of the generic collection supplies to be substituted with the generic type is known as the *type argument*. Constructed types that use different type arguments but are based on the same generic type aren't equivalent. For example, an object of type `GrowableArray<int>` couldn't be passed to a method if a `GrowableArray<string>` was expected. Constructed types can't be substituted even if an implicit conversion exists between the type arguments. The following code sample *won't* compile, despite the implicit cast available from `int` to `double` and `object`:

```
public void MethodOne(GrowableArray<double> gad)
{
 ;
}

public void MethodTwo(GrowableArray<object> gad)
{
 ;
}

static void main()
{
 GrowableArray<int> ga = new GrowableArray<int>();
 MethodOne(ga);  //will NOT compile
 MethodTwo(ga);  //will NOT compile
}
```

This restriction applies only to the constructed type, not to the type arguments. If an implicit conversion exists between the type of a variable and the type of the type argument, conversion will be successful as expected. This means the following code is legal:

```
GrowableArray<double> ga = new GrowableArray<double>();
int i = 7;
ga.AddElement(i); //the variable i is implicitly converted to a double
```

A generic type may contain any number of type parameters. An associative collection such as a dictionary or map, where a value is associated with a particular key, is a common use of multiple type parameters, because the type of the key and type of the value will typically be different. You should separate type parameters using commas within the angle brackets:

```
public class Dictionary<T, K> {}
```

You can declare a constructed type of a generic type with multiple type parameters by providing a comma-delimited list of type arguments:

```
Dictionary<string, int> dictionaryOfStringToInt = new Dictionary<string, int>();
```

Constraints

In all the examples up to this point, variables that are instances of type parameters haven't had methods called upon them and new instances haven't been constructed. For generic classes that are simple containers or that implement simple algorithms, this limitation is fine; however, for many real-world cases, variables of generic types need to be created and have methods called upon them inside the generic class. Casting the variable represented by a type parameter is an option, but this defeats the compile-time type-checking benefits that are such a large part of generics. To overcome this problem, type parameters can have one or more constraints that define the structural qualities a type must exhibit to be an eligible argument for a particular template parameter.

■ **Note** .NET generics differ from C++ templates with regard to constraints. In C++, templates are a compile-time concept, so the equivalent of constraints is unnecessary. The methods that a template type can call and the types that a developer is attempting to place in the template are available to the same compiler at the same time. In .NET, generics are a runtime concept, and a generic class written in one language can be used in another language.

Constraints can specify the following:

- Whether a type argument is a class or a struct
- A number of interfaces that the argument must implement
- A class that the type must be of or must derive from
- The existence of a default constructor

Each type parameter can have a different set of constraints, and all constraints are specified after the type parameter list using a `where` clause. In the following sample, both type parameters have constraints; the template parameter `T` is required to be a class that implements `IConvertible` and provides a default constructor, and `K` must be a struct:

```
public class Container<T, K>
 where T: class, IConvertible, new()
 where K: struct
{
  public Container()
  {
   T t = new T();
   int i = t.ToInt32(null);
   K k = new K();
  }
}
```

In the constructor, the type parameters declare local variables, and the variable of the generic type `T` has the `ToInt32` method of `IConvertible` called upon it. Type parameter `K` was constrained to be a `struct`, which means it will act as though it has a parameterless constructor, so `new` instances of `K` can be created without the `new` constraint being specified. It's an error to apply the `new` constraint to a `struct`, and a number of other invalid constraints aren't legal, such as the provision of multiple class constraints and the use of class constraints that specify a sealed class.

Constraints must appear in a certain order. If a type parameter is going to be constrained to a `class`, `struct`, or particular subclass, this constraint must appear first. Any number of interface constraints can then appear, and the final optional constraint is the constructor constraint.

Generic Methods

It's sometimes desirable for generics to be utilized at the method level rather than a type level. This could be the case for a class that exposes procedural methods or where the generic type is relevant only to a specific function. In these cases, you can define the generic template entirely at the function level, and the class that the generic method is defined in doesn't need to be generic:

```
public class Utils
{
 Random _random = new Random();

 public T ReturnRandomElementOfArray<T>(T[] coll)
 {
  return coll[_random.Next(coll.Length)];
 }
}
```

In this example, a utility class has a method that returns a random element of an array, and by using a generic method, you can realize the benefits of type-safety and avoid boxing. You can call the method by specifying the type argument in angled brackets after the method name; alternatively, you can omit the type argument, and the code can rely on compiler-provided inference to determine the value of the type argument. The following code shows both cases:

```
Utils u = new Utils();
double[] dblColl = new double[]{1.0, 2.0, 3.0};
int[] intColl = new int[]{4, 5, 6};

//type argument specified
double dblRandom = u.ReturnRandomElementOfArray<double>(dblColl);

//type argument inferred
int intRandom = u.ReturnRandomElementOfArray(intColl);
```

Relying on type argument inference may make the calling code become more compact and appear more natural, but you can receive some unexpected consequences in the face of overloading and version changes. If you updated the `Utils` class shown in the previous example to include a nongeneric version of `ReturnRandomElementOfArray` that took an `int` array and returned a single `int`, the code in the previous sample that relies on type argument inference would switch to the nongeneric method when recompiled against the newer version. Although this is technically the correct behavior on the part of the compiler, it may surprise you to have a different method called in the absence of modifications in your source code. It's therefore a good idea to avoid overloading generic and nongeneric methods when possible.

You can apply the same constraints to generic methods as you do to nongeneric methods. When a generic method is overridden in a derived class, any constraints on the type parameters are inherited as well and can't be repeated in the overridden method. The following code illustrates a number of these scenarios. Note that the type parameter name doesn't need to be constant and can be changed in derived classes. Unless a class is inheriting a generic class and implementing a generic interface (covered in a moment), changing the name of a type parameter isn't recommended.

```
//generic base class with a generic method
public class GenericBase<T>
{
 public virtual void MyMethodUsingGenericParameter(T t) { }
 public virtual void MyGenericMethod<W>(W w) where W: IComparable{ }
}

//derived generic class
public class GenericInherited<V>: GenericBase<V>
{
 public override void MyMethodUsingGenericParameter(V v) { }
 public override void MyGenericMethod<W>(W w) { }  //IComparable constraint
 //is inherited from GenericBase
}

//nongeneric class
public class NonGenericInherited : GenericInherited<int>
{
 public override void MyMethodUsingGenericParameter(int i) { }
 public override void MyGenericMethod<W>(W w) { }
}
```

Inheritance, Overriding, and Overloading

Generics and inheritance can mix in interesting ways. The first point to note is that a generic parameter *can't* act as the base class for a generic type (regardless of any constraints such as `class`), meaning the following code isn't legal:

```
public class WillNotCompile<T> : T { }
// error CS0689: Cannot derive from 'T' because it is a type parameter
```

This prohibits the popular "mix-in" style of generic programming. Although this capability isn't allowed in C# 2.0, the C# design team understands the utility of such constructs, and it may appear in future versions of the language.

Other than this exclusion, inheritance and generics have few scenarios that aren't permitted. All of the following scenarios are legal in C#:

- A generic class can have a nongeneric class as a base.
- A nongeneric class can have a constructed generic class as a base if all type arguments are provided.
- A generic class can have a generic class as a base. If there are constraints on the type parameters in the base class, these must be repeated with the derived class; the requirement to repeat the constraints is in contrast to the implicit inheritance of method-level constraints.
- If a generic class has multiple generic parameters, a base class can provide type arguments for zero or more of the type parameters.

The following sample demonstrates the first two scenarios:

```
//base - a nongeneric class
public class BaseClass { }

//first tier - a derived generic class
public class DerivedGenericClass<T>: BaseClass {}

//second tier - a derived generic class and a derived nongeneric class
public class DerivedClass: DerivedGenericClass<int> {}
public class SecondDerivedGenericClass <T>: DerivedGenericClass<T> { }
```

The following sample shows the third scenario. If the constraint isn't included in the derived generic class, it isn't legal.

```
//constraints repeated in derived generic class
public class GenericBase<T> where T : new() {}
public class GenericDerived<T> : GenericBase<T> where T : new() { }
```

Finally, the following code shows the fourth scenario where a nongeneric type is derived from a generic type in two inheritance steps:

```
public class Dictionary<T, K> { }
public class StringDictionary<K> : Dictionary<string, K> { }
public class StringToIntDictionary: StringDictionary<int> { }
public class StringToDoubleDictionary : Dictionary<string, double> { }
```

Generic Interfaces, Delegates, and Events

Interfaces can have type parameters in much the same way as classes. The only major restriction is that a particular generic interface can appear only once in a particular class hierarchy, which means a class can't implement the same generic interface twice with a different type argument. The following code shows permitted interface implementation scenarios, with the invalid implementation commented out:

```
public interface IGeneric<T>
{
 T InterfaceMethod();
}

public class GenericInterfaceImplementor<T>: IGeneric<T>
{
 public T InterfaceMethod() { return default(T); }
}

public class InterfaceImplementor: IGeneric<int>
{
```

```
 public int InterfaceMethod() { return 0; }
}

/* compile error CS0111: Type 'DerivedInterfaceImplementor' already
defines a member called
'InterfaceMethod' with the same parameter types

public class DerivedInterfaceImplementor : InterfaceImplementor, IGeneric<double>
{
 public int InterfaceMethod() { return 0; }
 public double InterfaceMethod() { return 0.0; }
}
*/
```

Note the use of the `default` keyword in `GenericInterfaceImplementor.InterfaceMethod`. You can use the `default` keyword to assign or compare an instance of a parameterized type without needing to know or specify whether the parameterized type is a class or a struct. For classes, `default` is the same as null and the default value for value types. In the case of structs, the member variables of a `default` instance follow the rules for classes and primitives.

Delegates support generics using a pattern similar to generic methods. A delegate can use a type parameter in a generic class, or a generic delegate can be declared within a nongeneric class. In the case of the latter, you can express constraints with the delegate declaration:

```
public class GenericClass <T>
{
 public delegate T GenericDelegate(int i);

 public void UseDelegate(GenericDelegate del) {}
}

public class NonGeneric
{
 public delegate T GenericDelegate1<T>(int i) where T : class;
 public delegate U GenericDelegate2<U, V>(V v);

 public void UseDelegate<T>(GenericDelegate1<T> del) where T : class { }
}
```

In the example, the type parameter and constraint for `T` appears in `GenericDelegate1` and `UseDelegate`. Notice that when a generic delegate is declared on a nongeneric class, the type parameters and constraints must be repeated in every declaration that uses the type parameter. This is the same as the situation that exists with generic methods on a nongeneric class. The only case in which the constraint doesn't need to be repeated is with overridden generic methods.

C# allows the full definition of type arguments when an instance of a delegate is declared, or the programmer can rely on the compiler to infer type arguments. Using the delegate methods shown in the previous sample, all the following forms of delegate instantiation are supported:

```
string StandardMethod(int i) { return ""; }

public void UseGenericDelegates()
{
 GenericClass<string> gcs = new GenericClass<string>();

 //generic method can be called used explicitly like this
 GenericClass<string>.GenericDelegate del = new
  GenericClass<string>.GenericDelegate(StandardMethod);
 gcs.UseDelegate(del);

 //or implicitly like this
 gcs.UseDelegate(StandardMethod);

 //implicit use of generic delegate
 NonGeneric ng = new NonGeneric();
 ng.UseDelegate<string>(StandardMethod);
```

For a delegate declared outside a class, the option of using the containing class's type parameters is obviously unavailable, and type parameters and any constraints must be declared with the delegate.

You can associate a generic delegate with events. This can be a particularly useful technique, as it allows a single delegate to be associated with multiple events. Each event can specify custom type arguments, allowing strong typing to be achieved without the need to specify a new delegate signature. The following sample shows a generic delegate that's associated with an event; this raises events that provide information about the time that event was raised in the `EventArgs`-derived argument:

```
//generic delegate that takes sender and event args type parameters
 public delegate void StandaloneGenericDelegate<S, A>(S sender, A args);


 public class TimedEventArgs: EventArgs
 {
  //constructor
  public TimedEventArgs(DateTime time) { _timeOfEvent = time; }

  //member variable
  DateTime _timeOfEvent;

  //property
  public DateTime TimeOfEvent { get{ return _timeOfEvent; }}

 }
```

```
 public class TimedEventRaiser
 {
public event StandaloneGenericDelegate<TimedEventRaiser, TimedEventArgs>
 TimedEvent;

  public void Raiser()
  {
   if (TimedEvent != null)
   {
    TimedEvent(this, new TimedEventArgs(DateTime.Now));
   }
  }
 }
```

Conclusion and Design Guidance

Generics are a welcome addition to the C# language. They provide two key features: compile-time type-safety and the elimination of boxing and unboxing operations for value types. Generics fall into the group of language elements that initially appear rather bland, utilitarian, and limited in scope, but they allow powerful code patterns and elegant designs that are difficult to achieve in their absence.

The real design clue that indicates generics are required is loosely typed declarations. If parameters or return types are declared as `object` even though it feels like stronger typing would be better, it's generally a good time to introduce generics. Feeling a desire for stronger typing is often associated with the requirement to perform a significant number of casting operations. So, to make this advice a bit more concrete, if there's a couple of casts (or more) required to perform a particular logical operation, such as opening a database connection, it's a good indication you should use generics.

As with any new language or developer tool feature, there can be a tendency to overuse the "new toy" to solve every coding problem. And as with all new toys, the greatest joy and satisfaction may come from using the toy in unanticipated and novel ways, but be wary of overuse that can come from the "everything looks like a nail when my only tool is a hammer" pattern of thought.

A common question about generic constraints is, why are they so limited in what they support?

Although it's possible to provide a more extensive constraint syntax, it's not clear at this point where the ultimate "sweet spot" is for such support, so it was decided to "step lightly" in this version, with the possibility for more support in future versions of the language.

CHAPTER 18

Strings

All strings in C# are instances of the `System.String` type in the CLR. Because of this, many built-in operations are available that work with strings. For example, the `String` class defines an indexer function that can be used to iterate over the characters of the string:

```
using System;
class Test
{
    public static void Main()
    {
        string s = "Test String";

        for (int index = 0; index < s.Length; index++)
            Console.WriteLine("Char: {0}", s[index]);
    }
}
```

Operations

The `string` class is an example of an immutable type, which means the characters contained in the string can't be modified by users of the string. All modifying operations on a `string` return a new string instance rather than modifying the instance on which the method is called.

Immutable types are used to make reference types that have value semantics (in other words, act somewhat like value types).

The `string` class supports the comparison and searching methods listed in Table 18-1.

Table 18-1. *`string` Comparison and Search Methods*

Item	Description
`Compare()`	Compares two strings
`CompareOrdinal()`	Compares two string regions using an ordinal comparison
`CompareTo()`	Compares the current instance with another instance
`EndsWith()`	Determines whether a substring exists at the end of a string

Table 18-1. *(Continued)*

Item	Description
`StartsWith()`	Determines whether a substring exists at the beginning of a string
`IndexOf()`	Returns the position of the first occurrence of a substring
`LastIndexOf()`	Returns the position of the last occurrence of a substring

The `string` class supports the modification methods described in Table 18-2, which all return a new string instance.

Table 18-2. *`string` Modification Methods*

Item	Description
`Concat()`	Concatenates two or more strings or objects together. If objects are passed, the `ToString()` function is called on them.
`CopyTo()`	Copies a specified number of characters from a location in this string into an array.
`Insert()`	Returns a new string with a substring inserted at a specific location.
`Join()`	Joins an array of strings together with a separator between each array element.
`PadLeft()`	Right-aligns a string in a field.
`PadRight()`	Left-aligns a string in a field.
`Remove()`	Deletes characters from a string.
`Replace()`	Replaces all instances of a character with a different character.
`Split()`	Creates an array of strings by splitting a string at any occurrence of one or more characters.
`Substrng()`	Extracts a substring from a string.
`ToLower()`	Returns a lowercase version of a string.
`ToUpper()`	Returns an uppercase version of a string.
`Trim()`	Removes whitespace from a string.
`TrimEnd()`	Removes a string of characters from the end of a string.
`TrimStart()`	Removes a string of characters from the beginning of a string.

String Encodings and Conversions

From the C# perspective, strings are always Unicode strings. When dealing only in the .NET world, this greatly simplifies working with strings.

Unfortunately, it's sometimes necessary to deal with the messy details of other kinds of strings, especially when dealing with text files produced by older applications. The `System.Text`

namespace contains classes that can be used to convert between an array of bytes and a character encoding such as ASCII, Unicode, UTF7, or UTF8. Each coding is encapsulated in a class such as `ASCIIEncoding`.

To convert from a string to a block of bytes, the `GetEncoder()` method on the encoding class is called to obtain an `Encoder`, which is then used to do the encoding. Similarly, to convert from a block of bytes to a specific encoding, `GetDecoder()` is called to obtain a decoder.

Converting Objects to Strings

The function `object.ToString()` is overridden by the built-in types to provide an easy way of converting from a value to a string representation of that value. Calling `ToString()` produces the default representation of a value; you can obtain a different representation by calling `String.Format()`. See Chapter 34 for more information.

An Example

You can use the `Split` function to break a string into substrings at separators:

```
using System;
class Test
{
    public static void Main()
    {
        string s = "Oh, I hadn't thought of that";
        char[] separators = new char[] {' ', ','};
        foreach (string sub in s.Split(separators))
        {
            Console.WriteLine("Word: {0}", sub);
        }
    }
}
```

This example produces the following output:

```
Word: Oh
Word:
Word: I
Word: hadn't
Word: thought
Word: of
Word: that
```

The `separators` character array defines what characters the string will be broken on. The `Split()`function returns an array of strings, and the `foreach` statement iterates over the array and prints it out.

In this case, the output isn't particularly useful because the "," string gets broken twice. You can fix this by using the regular expression classes.

StringBuilder

Though you can use the `String.Format()` function to create a string based on the values of other strings, it isn't necessarily the most efficient way to assemble strings. The runtime provides the `StringBuilder` class to make this process easier.

The `StringBuilder` class supports the properties and methods described in Table 18-3 and Table 18-4.

Table 18-3. *`StringBuilder` Properties*

Property	Description
`Capacity`	Retrieves or sets the number of characters the `StringBuilder` can hold.
`[]`	The `StringBuilder` indexer is used to get or set a character at a specific position.
`Length`	Retrieves or sets the length.
`MaxCapacity`	Retrieves the maximum capacity of the `StringBuilder`.

Table 18-4. *`StringBuilder` Methods*

Method	Description
`Append()`	Appends the string representation of an object
`AppendFormat()`	Appends a string representation of an object, using a specific format string for the object
`EnsureCapacity()`	Ensures the `StringBuilder` has enough room for a specific number of characters
`Insert()`	Inserts the string representation of a specified object at a specified position
`Remove()`	Removes the specified characters
`Replace()`	Replaces all instances of a character with a new character

The following example demonstrates how you can use the `StringBuilder` class to create a string from separate strings:

```
using System;
using System.Text;
class Test
{
```

```
    public static void Main()
    {
        string s = "I will not buy this record, it is scratched";
        char[] separators = new char[] {' ', ','};
        StringBuilder sb = new StringBuilder();
        int number = 1;
        foreach (string sub in s.Split(separators))
        {
            sb.AppendFormat("{0}: {1} ", number++, sub);
        }
        Console.WriteLine("{0}", sb);
    }
}
```

This code will create a string with numbered words and will produce the following output:

```
1: I 2: will 3: not 4: buy 5: this 6: record 7:  8: it 9: is 10: scratched
```

Because the call to `split()` specified both the space and the comma as separators, it considers there to be a word between the comma and the following space, which results in an empty entry.

Regular Expressions

If the searching functions found in the `string` class aren't powerful enough, the `System.Text` namespace contains a regular expression class named `Regex`. Regular expressions provide a powerful method for doing search and/or replace functions.

Although this section contains a few examples of using regular expressions, a detailed description of them is beyond the scope of the book. Several regular expression books are available, and the subject is also covered in most books about Perl. *Mastering Regular Expressions*, Second Edition (O'Reilly, 2002), by Jeffrey E. F. Friedl, and *Regular Expression Recipes: A Problem-Solution Approach* (Apress, 2005), by Nathan A. Good, are two great references.

The regular expression class uses a rather interesting technique to get maximum performance. Rather than interpret the regular expression for each match, it writes a short program on the fly to implement the regular expression match, and that code is then run.[1]

You can revise the previous example of `Split()` to use a regular expression, rather than single characters, to specify how the split should occur. This will remove the blank word that was found in the preceding example.

1. The program is written using the .NET intermediate language—the same one that C# produces as output from a compilation. See Chapter 32 for information on how this works.

```
// file: regex.cs
using System;
using System.Text.RegularExpressions;
class Test
{
    public static void Main()
    {
        string s = "Oh, I hadn't thought of that";
        Regex regex = new Regex(@" |, ");
        char[] separators = {' ', ','};
        foreach (string sub in regex.Split(s))
        {
            Console.WriteLine("Word: {0}", sub);
        }
    }
}
```

This example produces the following output:

```
Word: Oh
Word: I
Word: hadn't
Word: thought
Word: of
Word: that
```

In the regular expression, the string is split either on a space or on a comma followed by a space.

Regular Expression Options

When creating a regular expression, you can specify several options to control how the matches are performed. `Compiled` is especially useful to speed up searches that use the same regular expression multiple times. Table 18-5 lists the regular expression options.

Table 18-5. *Regular Expression Options*

Option	Description
`Compiled`	Compiles the regular expression into a custom implementation so matches are faster
`ExplicitCapture`	Specifies that only valid captures are named
`IgnoreCase`	Performs case-insensitive matching
`IgnorePatternWhitespace`	Removes unescaped whitespace from the pattern to allow # comments

Option	Description
`Multiline`	Changes the meaning of the caret (^) and question mark (?) so they match at the beginning or end of any line, not the beginning or end of the whole string
`MultilineRightToLeft`	Performs searches from right to left rather than from left to right
`Singleline`	Single-line mode, where a dot (`.`) matches any character including `\n`

More Complex Parsing

Using regular expressions to improve the function of `Split()` doesn't really demonstrate their power. The following example uses regular expressions to parse an IIS log file. That log file looks something like this:

```
#Software: Microsoft Internet Information Server 4.0
#Version: 1.0
#Date: 1999-12-31 00:01:22
#Fields: time c-ip cs-method cs-uri-stem sc-status
00:01:31 157.56.214.169 GET /Default.htm 304
00:02:55 157.56.214.169 GET /docs/project/overview.htm 200
```

The following code will parse this into a more useful form:

```
// file=logparse.cs
// compile with: csc logparse.cs
using System;
using System.Net;
using System.IO;
using System.Text.RegularExpressions;
using System.Collections;

class Test
{
    public static void Main(string[] args)
    {
        if (args.Length  == 0) //we need a file to parse
        {
            Console.WriteLine("No log file specified.");
        }
        else
            ParseLogFile(args[0]);
    }
```

```
public static void ParseLogFile(string    filename)
{
    if (!System.IO.File.Exists(filename))
    {
        Console.WriteLine ("The file specified does not exist.");
    }
    else
    {
        FileStream f = new FileStream(filename, FileMode.Open);
        StreamReader stream = new StreamReader(f);

        string line;
        line = stream.ReadLine();    // header line
        line = stream.ReadLine();    // version line
        line = stream.ReadLine();    // Date line

        Regex    regexDate= new Regex(@"\:\s(?<date>[^\s]+)\s");
        Match    match = regexDate.Match(line);
        string    date = "";
        if (match.Length != 0)
            date = match.Groups["date"].ToString();

        line = stream.ReadLine();    // Fields line

        Regex    regexLine =
            new Regex(        // match digit or :
                    @"(?<time>(\d|\:)+)\s" +
                        // match digit or .
                    @"(?<ip>(\d|\.)+)\s" +
                        // match any nonwhite
                    @"(?<method>\S+)\s" +
                        // match any nonwhite
                    @"(?<uri>\S+)\s" +
                        // match any nonwhite
                    @"(?<status>\d+)");

            // read through the lines, add an
            // IISLogRow for each line
        while ((line = stream.ReadLine()) != null)
        {
            //Console.WriteLine(line);
            match = regexLine.Match(line);
```

```
                if (match.Length != 0)
                {
                    Console.WriteLine("date: {0} {1}", date,
                                      match.Groups["time"]);
                    Console.WriteLine("IP Address: {0}",
                                     match.Groups["ip"]);
                    Console.WriteLine("Method: {0}",
                                      match.Groups["method"]);
                    Console.WriteLine("Status: {0}",
                                      match.Groups["status"]);
                    Console.WriteLine("URI: {0}\n",
                                      match.Groups["uri"]);
                }
            }
            f.Close();
        }
    }
}
```

The general structure of this code should be familiar. This example has two regular expressions. The date string and the regular expression used to match it are as follows:

```
#Date: 1999-12-31 00:01:22
\:\s(?<date>[^\s]+)\s
```

In the code, regular expressions are usually written using the verbatim string syntax, since the regular expression syntax also uses the backslash character. Regular expressions are most easily read if they're broken down into separate elements. The following code matches the colon (`:`):

```
\:
```

The backslash (`\`) is required because the colon by itself means something else. The following code matches a single character of whitespace (a tab or space):

```
\s
```

In this next part, the `?<date>` names the value that will be matched so it can be extracted later:

```
(?<date>[^\s]+)
```

The `[^\s]` is called a *character group*, with the `^` character meaning "none of the following characters." This group therefore matches any nonwhitespace character. Finally, the `+` character means to match one or more occurrences of the previous description (nonwhitespace). The parentheses delimit how to match the extracted string. In the preceding example, this part of the expression matches `1999-12-31`.

To match more carefully, you could use the \d (digit) specifier, with the whole expression written as follows:

```
\:\s(?<date>\d\d\d\d-\d\d-\d\d)\s
```

That covers the simple regular expression. You can use a more complex regular expression to match each line of the log file. Because of the regularity of the line, we could have used Split(), but that wouldn't have been as illustrative. The clauses of the regular expression are as follows:

```
(?<time>(\d|\:)+)\s      // match digit or : to extract time
(?<ip>(\d|\.)+)\s        // match digit or . to get IP address
(?<method>\S+)\s         // any nonwhitespace for method
(?<uri>\S+)\s            // any nonwhitespace for uri
(?<status>\d+)           // any digit for status
```

Secure String

Confidential data is typically stored in two main data types—numeric types such as floats and integers will hold keys and initialization vectors used in encryption processes, and strings will hold data such as passwords, credit card names, and confidential document fragments. Securing integral types is typically quite easy; they can be zeroed out as soon as they aren't needed, and even when they hold confidential data, identifying them using the memory window of a debugger or using a crash dump analysis tool is quite hard because they don't look any different to the millions of other bytes that live in a process's address space.

Strings are a little different; the immutability of System.String means it's impossible to clear the contents once a value has been stored in it, and when a string is modified (causing a new allocation) or moved about during garbage collection, multiple copies of the string's characters are left lying around the address space of the process. This immutability is a significant risk if an attacker can manage to exploit a vulnerability to begin reading the process's memory, attach a debugger to the process, or cause the process to crash and capture the resulting memory dump file. Recognizing a group of characters that form words is an easy process when scanning a large binary file (and can easily be automated), and this makes finding the confidential data that was stored using String a quick and simple process.

These risks have been deemed sufficient to warrant the introduction of a new type in .NET 2.0 that addresses these issues: SecureString. SecureString is a string type that's pinned and encrypted in memory, and it's mutable so that its contents can be cleared when it's no longer needed. To provide a simple, standard way to clear the contents of SecureString, you can implement the IDisposable interface (see Chapter 8 for details of IDisposable) and clear it at the end of a using block:

```
static System.Security.SecureString ReadSecretData()
{
 System.Security.SecureString s =
  new System.Security.SecureString();
 //read in secret data
 return s;
}
```

```
static void Main(string[] args)
{
 using (ReadSecretData())
 {
  // do required processing of data
 }// SecureString cleared here
 //  SecureString is now empty
}
```

The number of methods supported by `SecureString` is quite limited; there are a few methods for populating the string (`AppendChar()`, `InsertAt()`, and `SetAt()`) and a few more for clearing the string (`Clear()` and `RemoveAt()`). There's also a `MakeReadOnly()` method to prevent the contents from changing.

There are *no* direct methods for creating a `SecureString` from a `String` object or for going the other way. This is a deliberate omission, as converting from or to a `String` negates the security benefit of `SecureString`. The indirect conversion route from a `String` is through an overloaded constructor of `SecureString` that takes a `char*` parameter, and `String` conversion is possible through the `Marshal.SecureStringToGlobalAllocUni()` method:

```
//don't do this at home (or work)!!
static unsafe void ToAndFromString()
{
 string s = "Some data";
 fixed (char* pS = s)
 {
  using (SecureString ss = new SecureString(pS, s.Length))
  {
   //a few random modifications
   ss.AppendChar('!');
   ss.InsertAt(1, '_');
   ss.SetAt(0, ' ');
   ss.RemoveAt(3);

   //make read-only
   ss.MakeReadOnly();

   //convert back to a string
   IntPtr ssData = Marshal.SecureStringToGlobalAllocUni(ss);
   String newString = Marshal.PtrToStringUni(ssData);
   Marshal.FreeHGlobal(ssData);
  }
 }
}
```

The contents of `SecureString` are protected by the Windows Data Protection API (DPAPI), which is available only on Windows 2000 (SP3 and newer), Windows XP, and Windows Server 2003 and newer. Attempting to create a `SecureString` on older platforms will result in a security exception being raised.

The support for `SecureString` in other types of the .NET Framework libraries is poor, but there are plans to provide overloads for security-centric methods that allow a `SecureString` to be used instead of a `String` in future releases. The main use of `SecureString` in the 2.0 release of .NET will simply be the in-memory processing of confidential data. As data access and UI libraries are modified to support it, end-to-end string security will be possible.

CHAPTER 19

Properties

A few months ago, writing some code, we came up with a situation where one of the fields in a class (`Filename`) could be derived from another (`Name`). We therefore decided to use the property idiom (or *design pattern*) in C++ and wrote a `getFilename()` function for the field that was derived from the other. We then had to walk through all the code and replace the reference to the field with calls to `getFilename()`. This took a while, since the project was fairly big.

We also had to remember that when we wanted to get the filename, we had to call the `getFilename()` member function, rather than merely referring to the `Filename` member of the class. This made the model a bit tougher to grasp; instead of `Filename` just being a field, we had to remember that we were really calling a function whenever we needed to access it. Similarly, we needed to call a `setFilename()` function to set the value of filename.

That `getFilename()` and `setFilename()` are logically related to a single "virtual field" isn't obvious when looking at a class, especially when class members are listed alphabetically, as they usually are. The property pattern is good for the author of a class but a bit clunky for the user of the class.

C# adds properties as first-class citizens of the language. Properties appear to be fields to the user of a class, but they use a block of code (known as an *accessor*) to get the current value and set a new value. You can separate the user model (a field) from the implementation model (a member function), which reduces the amount of coupling between a class and the users of a class, leaving more flexibility in design and maintenance.

In the .NET runtime, properties are implemented using a naming pattern and a little bit of extra metadata linking the member functions to the property name. This allows properties to appear as properties in some languages and merely as member functions in other languages.

Properties are used heavily throughout the .NET Framework; in fact, there are few (if any) public fields.

Accessors

A property consists of a property declaration and either one or two blocks of code—known as *accessors*[1]—that handle getting or setting the property. Here's a simple example:

1. In some languages/idioms, a set accessor is also known as a *mutator*.

```
class Test
{
    private string name;

    public string Name
    {
        get
        {
            return name;
        }
        set
        {
            name = value;
        }
    }
}
```

This class declares a property called `Name` and defines both a getter and a setter for that property. The getter merely returns the value of the private variable, and the setter updates the internal variable through a special parameter named `value`. Whenever the setter is called, the variable value contains the value to which the property should be set. The type of `value` is the same as the type of the property.

Properties can have a getter, a setter, or both. A property that has only a getter is called a *read-only* property, and a property that has only a setter is called a *write-only* property.

Properties and Inheritance

Like member functions, properties can also be declared using the `virtual`, `override`, or `abstract` modifiers. These modifiers are placed on the property and affect both accessors.

When a derived class declares a property with the same name as in the base class, it hides the entire property; it isn't possible to hide only a getter or setter.

Use of Properties

Properties separate the interface of a class from the implementation of a class. This is useful when the property is derived from other fields and when you want to do lazy initialization and fetch a value only if the user really needs it.

Suppose that a carmaker wanted to be able to produce a report that listed some current information about the production of cars:

```
using System;
class Auto
{
    public Auto(int id, string name)
    {
        this.id = id;
        this.name = name;
    }

        // query to find # produced
    public int ProductionCount
    {
        get
        {
            if (productionCount == -1)
            {
                // fetch count from database here.
            }
            return(productionCount);
        }
    }
    public int SalesCount
    {
        get
        {
            if (salesCount == -1)
            {
                // query each dealership for data
            }
            return(salesCount);
        }
    }
    string name;
    int id;
    int productionCount = -1;
    int salesCount = -1;
}
```

Both the `ProductionCount` and `SalesCount` properties are initialized to `-1`, and the expensive operation of calculating them is deferred until it's actually needed.

Side Effects When Setting Values

Properties are also useful to do something beyond merely setting a value when the setter is called. A shopping basket could update the total when the user changed an item count, for example:

```
using System;
using System.Collections;
class Basket
{
    internal void UpdateTotal()
    {
        total = 0;
        foreach (BasketItem item in items)
        {
            total += item.Total;
        }
    }

    ArrayList    items = new ArrayList();
    Decimal    total;
}
class BasketItem
{
    BasketItem(Basket basket)
    {
        this.basket = basket;
    }
    public int Quantity
    {
        get
        {
            return(quantity);
        }
        set
        {
            quantity = value;
            basket.UpdateTotal();
        }
    }
    public Decimal Price
    {
        get
        {
            return(price);
        }
```

```
        set
        {
            price = value;
            basket.UpdateTotal();
        }
    }
    public Decimal Total
    {
        get
        {
                // volume discount; 10% if 10 or more are purchased
            if (quantity >= 10)
                return(quantity * price * 0.90m);
            else
                return(quantity * price);
        }
    }

    int        quantity;     // count of the item
    Decimal    price;        // price of the item
    Basket     basket;       // reference back to the basket
}
```

In this example, the `Basket` class contains an array of `BasketItem`. When the price or quantity of an item is updated, an update is fired back to the `Basket` class, and the basket walks through all the items to update the total for the basket.

You could also implement this interaction more generally using events, which are covered in Chapter 24.

Static Properties

In addition to member properties, C# also allows the definition of static properties, which belong to the whole class rather than to a specific instance of the class. Like static member functions, static properties can't be declared with the `virtual`, `abstract`, or `override` modifiers.

When we discussed `readonly` fields in Chapter 8, we showed one case that initialized some static `readonly` fields. You can do the same thing with static properties without having to initialize the fields until necessary. The value can also be fabricated when needed and not stored. If creating the field is costly and it will likely be used again, then the value should be cached in a private field. If it's cheap to create or it's unlikely to be used again, it can be created as needed:

```
class Color
{
    public Color(int red, int green, int blue)
    {
        this.red = red;
        this.green = green;
        this.blue = blue;
    }

    int    red;
    int    green;
    int    blue;

    public static Color Red
    {
        get
        {
            return(new Color(255, 0, 0));
        }
    }
    public static Color Green
    {
        get
        {
            return(new Color(0, 255, 0));
        }
    }
    public static Color Blue
    {
        get
        {
            return(new Color(0, 0, 255));
        }
    }
}
class Test
{
    static void Main()
    {
        Color background = Color.Red;
    }
}
```

When the user wants one of the predefined color values, the getter in the property creates an instance with the proper color on the fly and returns that instance.

Property Efficiency

Returning to the first example in this chapter, consider the efficiency of the code when executed:

```
class Test
{
    private string    name;

    public string Name
    {
        get
        {
            return name;
        }
        set
        {
            name = value;
        }
    }
}
```

This may seem to be an inefficient design, because a member function call is added where you'd normally see a field access. However, there's no reason the underlying runtime environment can't inline the accessors as it would any other simple function, so there's often[2] no performance penalty in choosing a property instead of a simple field. The opportunity to be able to revise the implementation later without changing the interface can be invaluable, so properties are usually a better choice than fields for public members.

A small downside does remain when using properties; they aren't supported natively by all .NET languages, so other languages may have to call the accessor functions directly, which is a bit more complicated than using fields.

Property Accessibility

During the initial design of the C# language, the language designers thought that allowing different accessibility levels for get and set accessors was an unnecessary complexity. After a considerable amount of customer feedback, the designers decided to reverse this decision for C# 2.0.

The problem with a uniform accessibility level was that in many cases a public `get` accessor and an `internal`, `protected`, or `private` `set` accessor was required. C# 1.0 didn't support this, which resulted in code that looked something like this:

2. The Win32 version of the .NET runtime performs the inlining of trivial accessors, but other environments wouldn't have to do so.

```
class Test
{
 private string name;
 public string Name
 {
  get
  {
   return name;
  }
 }

 protected void SetName(string name)
 {
  this.name = name;
 }
}
```

In this sample, a pseudo-setter is declared outside the property declaration just to allow a different accessibility level to be defined. It was decided that this ugly work-around was worse than the added complexity that different accessibility can cause, and the property syntax was extended to allow accessibility levels for an accessor to be "overridden." Rewriting the `Test` class correctly, it becomes the following:

```
class Test
{
 private string name;
 public string Name
 {
  get
  {
   return name;
  }

  protected set
  {
   name = value;
  }
 }
}
```

Accessibility modifiers specified at the accessor level can restrict the accessibility level only below that specified at the property level. Therefore, if a property is declared as `protected`, it causes a compiler error to specify an accessor as having `public` visibility.

You can apply an accessibility modifier to either the `get` or the `set`, not to both. This means the most visible accessor must have that accessibility level specified at the property level, and a modifier can then be placed on either the `get` method or the `set` method—whichever has the stricter accessibility requirement.

Virtual Properties

If a property makes sense as part of the base class, it may make sense to make the property virtual. Virtual properties follow the same rules as other virtual entities. Here's a quick example of a virtual property:

```
using System;

public abstract class DrawingObject
{
    public abstract string Name
    {
        get;
    }
}
class Circle: DrawingObject
{
    string name = "Circle";

    public override string Name
    {
        get
        {
            return(name);
        }
    }
}
class Test
{
    public static void Main()
    {
        DrawingObject d = new Circle();
        Console.WriteLine("Name: {0}", d.Name);
    }
}
```

The abstract property `Name` is declared in the `DrawingObject` class, with an abstract `get` accessor. This accessor must then be overridden in the derived class.

When the `Name` property is accessed through a reference to the base class, the overridden property in the derived class is called.

CHAPTER 20

Indexers, Enumerators, and Iterators

It sometimes makes sense to be able to index an object like an array. You can do this by writing an indexer for the object, which can be thought of as a "smart" array. Just like a property looks like a field but has accessors to perform get and set operations, an indexer looks like an array but has accessors to perform array-indexing operations.

Indexing with an Integer Index

A class that contains a database row might implement an indexer to access the columns in the row:

```
using System;
using System.Collections;
class DataValue
{
    public DataValue(string name, object data)
    {
        this.name = name;
        this.data = data;
    }
    public string Name
    {
        get
        {
            return(name);
        }
        set
        {
            name = value;
        }
    }
```

```
    public object Data
    {
        get
        {
            return(data);
        }
        set
        {
            data = value;
        }
    }
    string    name;
    object data;
}
class DataRow
{
    public DataRow()
    {
        row = new ArrayList();
    }

    public void Load()
    {
        /* load code here */
        row.Add(new DataValue("Id", 5551212));
        row.Add(new DataValue("Name", "Fred"));
        row.Add(new DataValue("Salary", 2355.23m));
    }

        // the indexer (use -1 because column index is 1-based)
    public DataValue this[int column]
    {
        get
        {
            return((DataValue) row[column - 1]);
        }
        set
        {
            row[column - 1] = value;
        }
    }
    ArrayList    row;
}
```

```
class Test
{
    public static void Main()
    {
        DataRow row = new DataRow();
        row.Load();
        Console.WriteLine("Column 0: {0}", row[1].Data);
        row[1].Data = 12;    // set the ID
    }
}
```

The `DataRow` class has functions to load a row of data, has functions to save the data, and has an indexer function to provide access to the data. In a real class, the `Load()` function would load data from a database.

You write the indexer function the same way you write a property, except the indexer function takes an indexing parameter. You declare the indexer using the name `this` since it has no name.

Indexing with a String Index

A class can have more than one indexer. For the `DataRow` class, it might be useful to be able to use the name of the column for indexing:

```
using System;
using System.Collections;
class DataValue
{
    public DataValue(string name, object data)
    {
        this.name = name;
        this.data = data;
    }
    public string Name
    {
        get
        {
            return(name);
        }
        set
        {
            name = value;
        }
    }
```

```
    public object Data
    {
        get
        {
            return(data);
        }
        set
        {
            data = value;
        }
    }
    string    name;
    object data;
}
class DataRow
{
    public DataRow()
    {
        row = new ArrayList();
    }

    public void Load()
    {
        /* load code here */
        row.Add(new DataValue("Id", 5551212));
        row.Add(new DataValue("Name", "Fred"));
        row.Add(new DataValue("Salary", 2355.23m));
    }

    public DataValue this[int column]
    {
        get
        {
            return( (DataValue) row[column - 1]);
        }
        set
        {
            row[column - 1] = value;
        }
    }
```

```
    int FindColumn(string name)
    {
        for (int index = 0; index < row.Count; index++)
        {
            DataValue dataValue = (DataValue) row[index];
            if (dataValue.Name == name)
                return(index + 1);
        }
        return(-1);
    }
    public DataValue this[string name]
    {
        get
        {
            return( (DataValue) this[FindColumn(name)]);
        }
        set
        {
            this[FindColumn(name)] = value;
        }
    }
    ArrayList    row;
}
class Test
{
    public static void Main()
    {
        DataRow row = new DataRow();
        row.Load();
        DataValue val = row["Id"];
        Console.WriteLine("Id: {0}", val.Data);
        Console.WriteLine("Salary: {0}", row["Salary"].Data);
        row["Name"].Data = "Barney";    // set the name
        Console.WriteLine("Name: {0}", row["Name"].Data);
    }
}
```

The string indexer uses the `FindColumn()` function to find the index of the name and then uses the `int` indexer to do the proper thing.

Indexing with Multiple Parameters

Indexers can have more than one parameter to simulate a multidimensional virtual array. The following example illustrates such a use:

```
using System;

public class Player
{
    string name;

    public Player(string name)
    {
        this.name = name;
    }

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

public class Board
{
    Player[,] board = new Player[8, 8];

    int RowToIndex(string row)
    {
        string temp = row.ToUpper();
        return((int) temp[0] - (int) 'A');
    }

    int PositionToColumn(string pos)
    {
        return(pos[1] - '0' - 1);
    }

    public Player this[string row, int column]
    {
        get
        {
            return(board[RowToIndex(row), column - 1]);
        }
        set
        {
            board[RowToIndex(row), column - 1] = value;
        }
    }
```

```
    public Player this[string position]
    {
        get
        {
            return(board[RowToIndex(position),
                         PositionToColumn(position)]);
        }
        set
        {
            board[RowToIndex(position),
                      PositionToColumn(position)] = value;
        }
    }
}
class Test
{
    public static void Main()
    {
        Board board = new Board();

        board["A", 4] = new Player("White King");
        board["H", 4] = new Player("Black King");

        Console.WriteLine("A4 = {0}", board["A", 4]);
        Console.WriteLine("H4 = {0}", board["H4"]);
    }
}
```

The example implements a chessboard that can be accessed using standard chess notation (a letter from *A* to *H* followed by a number from 1 to 8). The first indexer accesses the board using string and integer indices, and the second indexer uses a single string such as "C5."

Enumerators and foreach

This section covers how to write enumerators by manually adding all the code that's required to implement the enumeration interfaces. C# 2.0 introduces new functionality called *iterators* that make writing enumerators much simpler. Many of the concepts of iterators rely on an understanding of how to implement enumeration, so you should read this section as background information rather than as a coding tutorial. We'll cover iterators later in this chapter.

If an object can be treated as an array, it's often convenient to iterate through the object using the `foreach` statement. To understand what's required to enable `foreach`, it helps to know what's happening behind the scenes.

When the compiler sees the following `foreach` block:

```
foreach (string s in myCollection)
{
    Console.WriteLine("String is {0}", s);
}
```

it transforms the code into the following:

```
IEnumerator enumerator = ((IEnumerable) myCollection).GetEnumerator();
while (enumerator. MoveNext())
{
    string s = (string) enumerator.Current();
    Console.WriteLine("String is {0}", s);
}
```

The first step of the process is to cast the collection class to `IEnumerable`. If that succeeds, the class supports enumeration, and an `IEnumerator` interface reference to perform the enumeration is returned. The `MoveNext()` and `Current` members of the class are then called to perform the iteration.

The `IEnumerator` interface can be implemented directly by the container class, or it can be implemented by a separate private class. Private implementation is preferable, since it simplifies the collection class and allows multiple users to iterate over the same instance at the same time.

The following example shows an integer collection class that enables `foreach` usage (note that this isn't intended to be a full implementation of such a class):

```
using System;
using System.Collections;

// Note: This class is not thread-safe
public class IntList: IEnumerable
{
    int[] values = new int[10];
    int allocated = values.Length;
    int count = 0;
    int revision = 0;

    public void Add(int value)
    {
            // reallocate if necessary...
        if (count + 1 == allocated)
        {
            int[] newValues = new int[allocated * 2];
            for (int index = 0; index < count; index++)
            {
                newValues[index] = values[index];
            }
            allocated *= 2;
        }
```

```
        values[count] = value;
        count++;
        revision++;
    }

    public int Count
    {
        get
        {
            return(count);
        }
    }

    void CheckIndex(int index)
    {
        if (index >= count)
            throw new ArgumentOutOfRangeException("Index value out of range");
    }

    public int this[int index]
    {
        get
        {
            CheckIndex(index);
            return(values[index]);
        }
        set
        {
            CheckIndex(index);
            values[index] = value;
            revision++;
        }
    }

    public IEnumerator GetEnumerator()
    {
        return(new IntListEnumerator(this));
    }

    internal int Revision
    {
        get
        {
            return(revision);
        }
    }
}
```

```
class IntListEnumerator: IEnumerator
{
    IntList    intList;
    int revision;
    int index;

    internal IntListEnumerator(IntList intList)
    {
        this.intList = intList;
        Reset();
    }

    public bool MoveNext()
    {
        index++;
        if (index >= intList.Count)
            return(false);
        else
            return(true);
    }

    public object Current
    {
        get
        {
            if (revision != intList.Revision)
                throw new InvalidOperationException
                ("Collection modified while enumerating.");
            return(intList[index]);
        }
    }

    public void Reset()
    {
        index = -1;
        revision = intList.Revision;
    }
}

class Test
{
    public static void Main()
    {
        IntList list = new IntList();
```

```
        list.Add(1);
        list.Add(55);
        list.Add(43);

        foreach (int value in list)
        {
            Console.WriteLine("Value = {0}", value);
        }

        foreach (int value in list)
        {
            Console.WriteLine("Value = {0}", value);
            list.Add(124);
        }
    }
}
```

The collection class itself needs only a couple of modifications. It implements `IEnumerable` and therefore has a `GetEnumerator()` method that returns an `IEnumerator` reference to an instance of the enumerator class that points to the current list.

The `IntListEnumerator` implements the enumeration on the `IntList` that it's passed using the `IEnumerator` interface and therefore implements the members of that interface.

Having a collection change as it's being iterated over is a bad thing, so these classes detect that condition (as illustrated in the second `foreach` in `Main()`). The `IntList` class has a revision number that it updates when the list contents change. The current revision number for the list is stored when the enumeration is started and then checked in the `Current` property to ensure that the list is unchanged.

Improving the Enumerator

The enumerator in the previous section has two deficiencies.

The first is that the enumerator isn't compile-time type-safe but only runtime type-safe. If you write the following code:

```
IntList intList = new IntList();
intList.Add(55);
//…
foreach (string s in intList)
{
}
```

the error can't be identified at compile time, but an exception will be generated when the code is executed. The reason that this can't be identified at compile time is that `IEnumerator.Current` is of type `object`, and in the previous example, converting from `object` to `int` is a legal operation.

A second problem with `Current` being of type `object` is that returning a value type (such as `int`) requires that the value type be boxed. This is wasteful, since `IntListEnumerator.Current` boxes the `int` only to have it immediately unboxed after the property is accessed.

To address this situation, the C# compiler implements a pattern-matching approach instead of a strict interface-based approach when dealing with enumerators. Instead of requiring the collection class to implement `IEnumerable`, it has to have a `GetEnumerator()` method. This method doesn't have to return `IEnumerator` but can return a real class instance for the enumerator. This enumerator, in turn, needs to have the usual enumerator functions (`MoveNext()`, `Reset()`, and `Current`), but the type of `Current` doesn't have to be `object`.

With this modification, a strongly typed collection class can now get compile-time type checking, and classes that store value types can avoid boxing overhead. The modifications to the classes are fairly simple. First, remove the interface names. Modify `IntList.GetEnumerator()` as follows:

```
public IntListEnumerator GetEnumerator()
{
    return(new IntListEnumerator(this));
}
```

Second, the modification to `IntListEnumerator.Current` is also minimal:

```
public int Current
{
    get
    {
        if (revision != intList.Revision)
            throw new InvalidOperationException
            ("Collection modified while
            enumerating.");
        return(intList[index]);
    }
}
```

That was easy.

Unfortunately, there's a problem. The standard method of enabling enumeration is to implement `IEnumerable` and `IEnumerator`, so any language that looks for those isn't going to be able to iterate over the `IntList` collection.

The solution is to add explicit implementations of those interfaces.[1] This means adding an explicit implementation of `IEnumerable.GetEnumerator()` to `IntList`:

```
IEnumerator IEnumerable.GetEnumerator()
{
    return(GetEnumerator());
}
```

It also means adding an explicit implementation of `IEnumerator.Current` to `IntListEnumerator`:

1. For more information, see Chapter 10.

```
object IEnumerator.Current
{
    get
    {
        return(Current);
    }
}
```

This now enables the standard method of iteration, and you can use the resulting class either with a compiler that supports the strongly typed pattern-matching approach or with a compiler that supports `IEnumerable/IEnumerator`.

Disposable Enumerators

Sometimes an enumerator holds a valuable resource, such as a database connection. The resources will be released when the enumerator is finalized, but it'd be useful if the resource could be released when the enumerator was no longer needed. Because of this, the expansion of a `foreach` isn't quite as simple as implied previously.

The C# compiler does this by relying on the `IDisposable` interface, in a similar manner to the `using` statement. It's a bit more complicated in this case, however. For the `using` statement, it's easy for the compiler to determine whether the class implements `IDisposable`, but that's not true in this case. The compiler must handle three cases when it expands:

```
foreach (Resource r in container) …
```

GetEnumerator() Returns IEnumerator

In this case, the compiler must determine dynamically whether the class implements `IDisposable`. The `foreach` expands to the following:

```
IEnumerator e = container.GetEnumerator();
try {
    while (e.MoveNext()) {
        Resource r = e.Current;
        ...;
    }
}
finally {
    IDisposable d = e as IDisposable;
    if (d != null) d.Dispose();
}
```

GetEnumerator() Returns a Class That Implements IDisposable

If the compiler can statically know that a class implements `IDisposable`, the compiler will call `Dispose()` without the dynamic test:

```
IEnumerator e = container.GetEnumerator();
try {
    while (e.MoveNext()) {
        Resource r = e.Current;
        ...;
    }
}
finally {
    ((IDisposable) e).Dispose();
}
```

GetEnumerator() Returns a Class That Doesn't Implement IDisposable

In this case, the normal expansion is used:

```
IEnumerator e = container.GetEnumerator();
while (e.MoveNext()) {
    Resource r = e.Current;
    ...;
}
```

Design Guidelines

You should use indexers only in situations where the abstraction makes sense. This usually depends on whether the object is a container for some other object.

VB .NET views what C# terms an *indexer* as a default property, and a class can have more than one indexed property in a VB .NET program. Since C# views an indexer as an indication that an object is composed of the indexed type, the VB .NET view doesn't map into the C# perspective (how can an object be an array of two different types?). C# therefore allows access to the default indexed property directly only.[2]

C# gives its indexer the name `Item`, which is fine from the C# perspective, because the name is never used. Languages such as VB .NET, however, do see the name, and it may therefore be helpful to set the name to something other than `Item`. You can do this by placing the `IndexerNameAttribute` on the indexer. (You can find this attribute in the `System.Runtime.CompilerServices` namespace.)

Finally, you can use iterators (covered next) to implement enumerable types. Iterators offload the tedious work of enumerator implementation to the compiler, reducing the chance of bugs and making types easier to code.

Iterators

Looking at the implementation of the `IntListEnumerator` class presented earlier in this chapter, it's apparent that a basic state machine is implemented that keeps track of the element in the collection that should be returned by the `Current` property, and the element counter that forms part of the state machine is advanced by the `MoveNext` method. The pattern of implementing an

2. If you have to access other ones, you can call the get and set functions directly.

IEnumerator-implementing object that enumerates over an array is well-established, and with a little assistance, the compiler would be able to produce the enumerator. This is the idea behind iterators, a new feature in C# 2.0 that significantly simplifies the process of writing enumerators.

You can implement an iterator using yield return statements, which return the values of a collection in an ordered manner. The introduction of yield as a new keyword may initially raise compatibility concerns for programmers with a large C# 1.0 code base that includes yield as a variable name, but this doesn't cause any compatibility problems, as yield is a keyword only in the context of a yield return statement; you can still use yield without the return as a variable name in C# 2.0.

If you rewrite the IntList collection presented earlier to use iterators, you get the following:

```
public class IntList : IEnumerable
{
 int[] values = new int[10];
 int count = 0;

 //other member variables
 public IEnumerator GetEnumerator()
 {
  for (int index = 0; index < count; ++index)
  {
   yield return values[index];
  }
 }
```

Building the enumerator using iterators considerably cuts down the code the developer needs to write. Under the covers, the compiler will generate a nested class that's functionally identical to IntListEnumerator presented earlier in this chapter, with the exception of the Reset method. Reset isn't an overly important method, as it's always possible to get an enumerator for a collection in its initial state by simply creating a new enumerator object. The code generated by an iterator will throw a NotSupportedException exception if Reset is called.

Iterators have no runtime support and are purely a C# compiler-provided feature.

By making it so easy to provide enumeration implementations, it's more likely the developer will provide more enumeration offerings than the typical full-collection forward-only enumerator. Providing a bidirectional enumerator that will support enumeration over a subset of a collection is simple to implement using iterators:

```
public class IntList : IEnumerable
{
 //other methods and members
 public IEnumerable BidirectionalSubrange(bool forward, int start, int end)
 {
  if (start < 0 || end >= count)
  {
   throw new IndexOutOfRangeException("Start must be zero or greater and" +
    " end must be less than the size of the collection");
  }
```

```
  int step = forward == true? 1: -1;
  for( int index = start; index != end; index += step )
  {
    yield return values[index];
  }
 }
}
```

You can use this method in C# by calling the `BidirectionalSubrange` method inside the `foreach` statement:

```
IntList il = new IntList();
il.Add(1);
il.Add(2);
il.Add(3);
il.Add(4);

foreach (int i in il.BidirectionalSubrange(false, 1, 3))
 Console.WriteLine(i);
```

Notice that `BidirectionalSubrange` returned an `IEnumerable` reference, as opposed to the `IEnumerator` reference returned by the `GetEnumerator` method.

It's possible to use multiple `yield return` statements within the one iterator. So, for example, you could enumerate the names of a person in the following way:

```
public class Person
{
 string firstName;
 string middleName;
 string lastName;

 public Person(string firstName, string middleName, string lastName)
 {
  this.firstName = firstName;
  this.middleName = middleName;
  this.lastName = lastName;
 }

 public IEnumerable Names()
 {
  yield return firstName;
  yield return middleName;
  yield return lastName;
 }
}
```

Complex Enumeration Patterns

Although the previous example was a little contrived, the ability to nest `yield return` statements makes writing enumerations over complex collections quite easy. Consider a hash table that allows multiple values to be stored against each key; this collection is useful for modeling many real-world relationships, such as keying each line item in an invoice from the invoice ID. Without iterators, it'd be quite difficult to build the state machine to provide an enumerator that returns a key, then each member of its value collection, followed by the next key, then each member of its value collection, and so on. Writing this enumerator with iterators is quite simple:

```
public class MutliMap
{
 Hashtable map = new Hashtable();

 public void Add(object key, object value)
 {
  if (map[key] == null)
   map[key] = new ArrayList();

  ((ArrayList)map[key]).Add(value);
 }

 public IEnumerator KeysAndValues()
 {
  foreach (object key in map.Keys)
  {
   yield return key;
   if (map[key] != null)
   {
    ArrayList values = (ArrayList)map[key];
    foreach (object value in values)
    {
     yield return value;
    }
   }
  }
 }
}
```

Notice the use of two separate `yield return` statements (highlighted with italics in the sample)—one for the key and another for each value.

To leave an iterator block, you can use the `yield break` statement. A `yield break` statement behaves in much the same way as a `return` statement, and if there's a `finally` block within the iterator, it will be executed as a result of `yield break`. From the perspective of the client code, a `yield break` will appear as if the enumeration is complete, and subsequent calls to `MoveNext` will return false.

Generic Enumeration

You can also use iterators to implement generic enumeration. The only change required to implement a generic iterator is to change the return type of the function from `IEnumerator` and `IEnumerable` to `IEnumerator<T>` and `IEnumerable<T>`. The actual iterator code remains unchanged. Converting the earlier `IntList` class into a generic equivalent, you get the following:

```
// Note: This class is not thread-safe
public class GenericList<T> : IEnumerable<T>
{
 T[] values = new T[10];
 int allocated = values.Length;
 int count = 0;
 int revision = 0;
 public void Add(T value)
 {
// reallocate if necessary...
  if (count + 1 == allocated)
  {
   T[] newValues = new T[allocated * 2];
   for (int index = 0; index < count; index++)
   {
    newValues[index] = values[index];
   }
   allocated *= 2;
  }
  values[count] = value;
  count++;
  revision++;
 }
 public int Count
 {
  get
  {
   return (count);
  }
 }
 void CheckIndex(int index)
 {
  if (index >= count)
   throw new ArgumentOutOfRangeException("Index value out of range");
 }
 public T this[int index]
 {
  get
  {
   CheckIndex(index);
   return (values[index]);
  }
```

```
  set
  {
   CheckIndex(index);
   values[index] = value;
   revision++;
  }
 }
 public IEnumerator<T> GetEnumerator()
 {
  for (int index = 0; index < count; ++index)
  {
   yield return values[index];
  }
 }

 public IEnumerable<T> BidirectionalSubrange(bool forward, int start, int end)
 {
  if (start < 0 || end >= count)
  {
   throw new IndexOutOfRangeException("Start must be zero or greater and end must" +
    " be less than the size of the collection");
  }

  if (forward)
  {
   for (int index = start; index < end; ++index)
   {
    yield return values[index];
   }
  }
  else
  {
   for (int index = end; index >= start; --index)
   {
    yield return values[index];
   }
  }
 }

 internal int Revision
 {
  get
  {
   return (revision);
  }
 }
}
```

`IEnumerator<T>` doesn't contain a `Reset` method, so the earlier discussion about `Reset` throwing a `NotSupportedException` exception doesn't apply.

Design Guidelines

Iterators make writing enumerators over collection and collection-like classes much easier. By offloading the implementation of the state machine needed to track the position in the collection that a particular enumeration is up to, iterators dramatically simplify the C# code needed to implement even the most complex enumerations.

As with all code simplification features, particularly those that are new, there's a tendency to initially overuse the feature. Although this isn't overly harmful in the case of iterators, you should strive to achieve a balance between offering a rich set of enumeration options and avoiding bogging a class down with an excess of enumerators that can make changing the internal implementation of the collection laborious.

CHAPTER 21

Enumerations

Enumerations are useful when a value in the program can have a specific set of values only. For example, when a control can be only one of four colors, or when a network package can support only two protocols, an enumeration can improve the code.

A Line-Style Enumeration

In the following example, a line-drawing class uses an enumeration to declare the styles of lines it can draw:

```
using System;
public class Draw
{
    public enum LineStyle
    {
        Solid,
        Dotted,
        DotDash,        // trailing comma is optional
    }

    public void DrawLine(int x1, int y1, int x2, int y2, LineStyle lineStyle)
    {
        switch (lineStyle)
        {
            case LineStyle.Solid:
                // draw solid
                break;

            case LineStyle.Dotted:
                // draw dotted
                break;
```

```
            case LineStyle.DotDash:
                // draw dotdash
                break;

            default:
                throw(new ArgumentException("Invalid line style"));
        }
    }
}
class Test
{
    public static void Main()
    {
        Draw draw = new Draw();
        draw.DrawLine(0, 0, 10, 10, Draw.LineStyle.Solid);
        draw.DrawLine(5, 6, 23, 3, (Draw.LineStyle) 35);
    }
}
```

The `LineStyle` enum defines the values that can be specified for the enum, and then that same enum is used in the function call to specify the type of line to draw.

Although enums prevent the accidental specification of values outside the enum range, the values that can be specified for an enum aren't limited to the identifiers specified in the enum declaration. The second call to `DrawLine()` is legal, so an enum value passed into a function must still be validated to ensure it's in the range of valid values. The `Draw` class throws an invalid argument exception if the argument is invalid.

Enumeration Base Types

Each enumeration has an underlying type that specifies how much storage is allocated for that enumeration. The valid base types for enumeration are `byte`, `sbyte`, `short`, `ushort`, `int`, `uint`, `long`, and `ulong`. If the base type isn't specified, the base type defaults to `int`. The base type is specified by listing the base type after the enum name:

```
enum SmallEnum : byte
{
    A,
    B,
    C,
    D
}
```

Specifying the base type can be useful if size is a concern or if the number of entries would exceed the number of possible values for `int`.

Initialization

By default, the value of the first enum member is set to 0 and incremented for each subsequent member. Specific values may be specified along with the member name:

```
enum Values
{
    A = 1,
    B = 5,
    C = 3,
    D = 42
}
```

Computed values can also be used, as long as they depend only on values already defined in the enum:

```
enum Values
{
    A = 1,
    B = 2,
    C = A + B,
    D = A * C + 33
}
```

If an enum is declared without a 0 value, this can lead to problems, since 0 is the default initialized value for the enum:

```
enum Values
{
    A = 1,
    B = 2,
    C = A + B,
    D = A * C + 33
}
class Test
{
    public static void Member(Values value)
    {
        // do some processing here
    }
    public static void Main()
    {
        Values value = 0;
        Member(value);
    }
}
```

A member with the value 0 should always be defined as part of an enum.

Bit Flag Enums

Enums may also be used as bit flags by specifying a different bit value for each bit. Here's a typical definition:

```
using System;
[Flags]
enum BitValues : uint
{
    NoBits = 0,
    Bit1 = 0x00000001,
    Bit2 = 0x00000002,
    Bit3 = 0x00000004,
    Bit4 = 0x00000008,
    Bit5 = 0x00000010,
    AllBits = 0xFFFFFFFF
}
class Test
{
    public static void Member(BitValues value)
    {
        // do some processing here
    }
    public static void Main()
    {
        Member(BitValues.Bit1 | BitValues.Bit2);
    }
}
```

The `[Flags]` attribute before the enum definition is used so that designers and browsers can present a different interface for enums that are flag enums. In such enums, it makes sense to allow the user to `OR` several bits together, which doesn't make sense for nonflag enums.

The `Main()` function `OR`s two bit values together and then passes the value to the member function.

Conversions

Enum types can be converted to their underlying type and back again with an explicit conversion:

```
enum Values
{
    A = 1,
    B = 5,
    C = 3,
    D = 42
}
```

```
class Test
{
    public static void Main()
    {
        Values v = (Values) 2;
        int ival = (int) v;
    }
}
```

The sole exception to this is that you can convert the literal `0` to an enum type without a cast. This is allowed so you can write the following code:

```
public void DoSomething(BitValues bv)
{
    if (bv == 0)
    {

    }
}
```

You have to write the `if` statement as follows if this exception isn't present:

```
if (bv == (BitValues) 0)
```

That's not bad for this example, but it could be quite cumbersome in actual use if the enum is nested deeply in the hierarchy, like so:

```
if (bv == (CornSoft.PlotLibrary.Drawing.LineStyle.BitValues) 0)
```

That's a lot of typing.

The System.Enum Type

Like the other predefined type, the `Enum` type has some methods that make enums in C# a fair bit more useful than enums in C++.

The first of these is that the `ToString()` function is overridden to return the textual name for an enum value so you can do the following:

```
using System;

enum Color
{
    red,
    green,
    yellow
}
```

```
public class Test
{
    public static void Main()
    {
        Color c = Color.red;

        Console.WriteLine("c is {0}", c);
    }
}
```

The example produces the following rather than merely giving the numeric equivalent of `Color.red`:

```
c is red
```

You can perform other operations as well:

```
using System;

enum Color
{
    red,
    green,
    yellow
}

public class Test
{
    public static void Main()
    {
        Color c = Color.red;

            // enum values and names
        foreach (int i in Enum.GetValues(c.GetType()))
        {
            Console.WriteLine("Value: {0} ({1})", i, Enum.GetName(c.GetType(), i));
        }

            // or just the names
        foreach (string s in Enum.GetNames(c.GetType()))
        {
            Console.WriteLine("Name: {0}", s);
        }
```

```
            // enum value from a string, ignore case
        c = (Color) Enum.Parse(typeof(Color), "Red", true);
        Console.WriteLine("string value is: {0}", c);

            // see if a specific value is a defined enum member
        bool defined = Enum.IsDefined(typeof(Color), 5);
        Console.WriteLine("5 is a defined value for Color: {0}", defined);    }
}
```

The output from this example is as follows:

```
Value: 0 (red)
Value: 1 (green)
Value: 2 (yellow)
Name: red
Name: green
Name: yellow
string value is: red
5 is a defined value for Color: False
```

In this example, the values and/or names of the enum constants can be fetched from the enum, and the string name for a value can be converted to an enum value. Finally, a value is checked to see if it's the same as one of the defined constants.

CHAPTER 22

Attributes

In most programming languages, some information is expressed through declarations, and other information is expressed through code. For example, in the following class member declaration, the compiler and runtime will reserve space for an integer variable and set its protection so it's visible everywhere:

```
public int Test;
```

This is an example of declarative information; it's nice because of the economy of expression and because the compiler handles the details for you.

Typically, the types of declarative information are predefined by the language designer, and they can't be extended by users of the language. A user who wants to associate a specific database field with a field of a class, for example, must invent a way of expressing that relationship in the language, a way of storing the relationship, and a way of accessing the information at runtime. In a language such as C++, the user might define a macro that stores the information in a field that's part of the object. Such schemes work, but they're error-prone and not generalized. They're also ugly.

The .NET runtime supports *attributes*, which are merely annotations placed on elements of source code, such as classes, members, parameters, and so on. You can use attributes to change the behavior of the runtime, provide transaction information about an object, or convey organizational information to a designer. The attribute information is stored with the metadata of the element and can be easily retrieved at runtime through a process known as *reflection*.

C# uses a conditional attribute to control when member functions are called. The following is an example of the conditional attribute:

```
using System.Diagnostics;
class Test
{
    [Conditional("DEBUG")]
    public void Validate()
    {
    }
}
```

Most programmers will use predefined attributes much more often than they will write an attribute class.

Using Attributes

Suppose you're doing a project with a group, and suppose it's important to keep track of the code reviews that have been performed on the classes so you can determine when you're done with them. You could store the code review information in a database, which would allow easy queries about status, or you could store it in comments, which would make it easy to look at the code and the information at the same time.

Or you could use an attribute, which would enable both kinds of access.

To do that, you need an attribute class. An attribute class defines the name of an attribute, how it can be created, and the information that will be stored. The "An Attribute of Your Own" section covers the gritty details of defining attribute classes.

The example attribute class looks like this:

```
using System;
[AttributeUsage(AttributeTargets.Class)]
public class CodeReviewAttribute: System.Attribute
{
    public CodeReviewAttribute(string reviewer, string date)
    {
        this.reviewer = reviewer;
        this.date = date;
    }
    public string Comment
    {
        get
        {
            return(comment);
        }
        set
        {
            comment = value;
        }
    }
    public string Date
    {
        get
        {
            return(date);
        }
    }
    public string Reviewer
    {
        get
        {
            return(reviewer);
        }
    }
```

```
    string reviewer;
    string date;
    string comment;
}
[CodeReview("Eric", "01-12-2000", Comment="Bitchin' Code")]
class Complex
{
}
```

The `AttributeUsage` attribute before the class specifies that this attribute can be placed only on classes. When an attribute is used on a program element, the compiler checks to see whether the use of that attribute on that program element is allowed.

The naming convention for attributes is to append `Attribute` to the end of the class name. This makes it easier to tell which classes are attribute classes and which classes are normal classes. All attributes must derive from `System.Attribute`.

The class defines a single constructor that takes a reviewer and a date as parameters, and it also has the public string property `Comment`.

When the compiler comes to the attribute used on the class `Complex`, it first looks for a class derived from `Attribute` named `CodeReview`. It doesn't find one, so it next looks for a class named `CodeReviewAttribute`, which it finds.

Next, it checks to see whether the attribute is allowed for this usage (on a class).

Then, it checks to see if there's a constructor that matches the parameters you've specified in the attribute use. If it finds one, an instance of the object is created—the constructor is called with the specified values.

If named parameters exist, it matches the name of the parameter with a field or property in the attribute class, and then it sets the field or property to the specified value.

After this is done, the current state of the attribute class is saved to the metadata for the program element for which it was specified.

At least, this is what happens logically. In actuality, it only looks like it happens this way; see the "Attribute Pickling" sidebar for a description of how it's implemented.

ATTRIBUTE PICKLING

It doesn't really work the way just described for a few reasons, which are related to performance. For the compiler to actually create the attribute object, the .NET runtime environment would have to be running, so every compilation would have to start up the environment, and every compiler would have to run as a managed executable.

Additionally, the object creation isn't really required, since you're just going to store the information away.

The compiler therefore validates that it *could* create the object, call the constructor, and set the values for any named parameters. The attribute parameters are then pickled into a chunk of binary information, which is tucked away with the metadata of the object.

A Few More Details

You can use some attributes only once on a given element. You can use others, known as *multiuse* attributes, more than once. They might be used, for example, to apply several different security attributes to a single class. The documentation on the attribute will describe whether an attribute is single-use or multiuse.

In most cases, it's clear that the attribute applies to a specific program element. However, consider the following case:

```
using System.Runtime.InteropServices;
class Test
{
    [return: MarshalAs(UnmanagedType.LPWStr)]
    public static extern string GetMessage();
}
```

In most cases, an attribute in that position would apply to the member function, but this attribute is really related to the return type. How can the compiler tell the difference?

This can happen in the following situations:

- Method vs. return value
- Event vs. field or property
- Delegate vs. return value
- Property vs. accessor vs. return value of getter vs. value parameter of setter

For each of these situations, one case is much more common than the other case, and it becomes the default case. To specify an attribute for the nondefault case, you must specify the element to which the attribute applies:

```
using System.Runtime.InteropServices;
class Test
{
    [return: MarshalAs(UnmanagedType.LPWStr)]
    public static extern string GetMessage();
}
```

The `return:` indicates that this attribute should be applied to the return value.

You can specify the element even if there's no ambiguity. Table 22-1 describes the identifiers.

Table 22-1. *Attribute Application Keywords*

Specifier	Description
`assembly`	The attribute is on the assembly.
`module`	The attribute is on the module.
`type`	The attribute is on a class or struct.

Specifier	Description
`method`	The attribute is on a method.
`property`	The attribute is on a property.
`event`	The attribute is on an event.
`field`	The attribute is on a field.
`param`	The attribute is on a parameter.
`return`	The attribute is on the return value.

Attributes that are applied to assemblies or modules must occur after any `using` clauses and before any code:

```
using System;
[assembly:CLSCompliant(true)]

class Test
{
    Test() {}
}
```

This example applies the `ClsCompliant` attribute to the entire assembly. All assembly-level attributes declared in any file that's in the assembly are grouped together and attached to the assembly.

To use a predefined attribute, start by finding the constructor that best matches the information to be conveyed. Next, write the attribute, passing parameters to the constructor. Finally, use the named parameter syntax to pass additional information that wasn't part of the constructor parameters.

For more examples of attribute use, refer to Chapter 33.

An Attribute of Your Own

To define attribute classes and reflect on them at runtime, you have to consider a few more issues. The following sections will discuss some things to consider when designing an attribute.

You have two major things to determine when writing an attribute. The first is the program elements that the attribute may be applied to, and the second is the information that will be stored by the attribute.

Attribute Usage

Placing the `AttributeUsage` attribute on an attribute class controls where the attribute can be used. The possible values for the attribute are listed in the `AttributeTargets` enumerator (see Table 22-2).

As part of the `AttributeUsage` attribute, one of these can be specified or a list of them can be `OR`ed together.

Table 22-2. *AttributeTargets Values*

Return Value	Meaning
`Assembly`	The program assembly
`Module`	The current program file
`Class`	A class
`Struct`	A struct
`Enum`	An enumerator
`Constructor`	A constructor
`Method`	A method (member function)
`Property`	A property
`Field`	A field
`Event`	An event
`Interface`	An interface
`Parameter`	A method parameter
`ReturnValue`	The method return value
`Delegate`	A delegate
`All`	Anywhere
`ClassMembers`	Class, struct, enum, constructor, method, property, field, event, delegate, interface

You can also use the `AttributeUsage` attribute to specify whether an attribute is single-use or multiuse. You do this with the named parameter `AllowMultiple`. Such an attribute looks like this:

```
[AttributeUsage(AttributeTargets.Method | AttributeTargets.Event,
                AllowMultiple = true)]
```

Attribute Parameters

The information the attribute will store should be divided into two groups: the information that's required for every use and the optional items.

The information that's required for every use should be obtained via the constructor for the attribute class. This forces the user to specify all the parameters when they use the attribute.

Optional items should be implemented as named parameters, which allows the user to specify whichever optional items are appropriate.

If an attribute has several different ways in which it can be created, with different required information, separate constructors can be declared for each usage. Don't use separate constructors as an alternative to optional items.

Attribute Parameter Types

The attribute pickling format supports only a subset of all the .NET runtime types, and therefore, you can use only some types as attribute parameters. The types allowed are the following:

- `bool`, `byte`, `char`, `double`, `float`, `int`, `long`, `short`, `string`
- `object`
- `System.Type`
- An `enum` that has public accessibility (not nested inside something nonpublic)
- A one-dimensional array of one of the previous types

Reflecting on Attributes

Once attributes are defined on some code, it's useful to be able to find the attribute values. You can do this through reflection.

The following code shows an attribute class, the application of the attribute to a class, and the reflection on the class to retrieve the attribute:

```
using System;
using System.Reflection;
[AttributeUsage(AttributeTargets.Class, AllowMultiple=true)]
public class CodeReviewAttribute: System.Attribute
{
    public CodeReviewAttribute(string reviewer, string date)
    {
        this.reviewer = reviewer;
        this.date = date;
    }
    public string Comment
    {
        get
        {
            return(comment);
        }
        set
        {
            comment = value;
        }
    }
```

```
    public string Date
    {
        get
        {
            return(date);
        }
    }
    public string Reviewer
    {
        get
        {
            return(reviewer);
        }
    }
    string reviewer;
    string date;
    string comment;
}
[CodeReview("Eric", "01-12-2000", Comment="Bitchin' Code")]
[CodeReview("Gurn", "01-01-2000", Comment="Revisit this section")]
class Complex
{
}

class Test
{
   public static void Main()
    {
        Type type = typeof(Complex);
        foreach (CodeReviewAttribute att in
                        type.GetCustomAttributes(typeof(CodeReviewAttribute), false))
        {
            CodeReviewAttribute    att = (CodeReviewAttribute) atts[0];
            Console.WriteLine("Reviewer: {0}", att.Reviewer);
            Console.WriteLine("Date: {0}", att.Date);
            Console.WriteLine("Comment: {0}", att.Comment);
        }
    }
}
```

The `Main()` function first gets the type object associated with the type `Complex`. It then iterates over all the `CodeReviewAttribute` attributes attached to the type and writes the values.

Alternately, the code could get all the attributes by omitting the type in the call to `GetCustomAttributes`:

```
foreach (object o in type.GetCustomAttributes(false))
{
    CodeReviewAttribute att = o as CodeReviewAttribute;
    if (att != null)
    {
        // write values here…
    }
}
```

This example produces the following output:

```
Reviewer: Eric
Date: 01-12-2000
Comment: Bitchin' Code
Reviewer: Gurn
Date: 01-01-2000
Comment: Revisit this section
```

The `false` value in the call to `GetCustomAttributes` tells the runtime to ignore any inherited attributes. In this case, it'll ignore any attributes on the base class of `Complex`.

In the example, you can obtain the type object for the `Complex` type using `typeof`. You can also obtain it in the following manner:

```
Complex c = new Complex();
Type t = c.GetType();
Type t2 = Type.GetType("Complex");
```

CHAPTER 23

Delegates and Anonymous Methods

Delegates are similar to interfaces in that they specify a contract between a caller and an implementer. Rather than specifying an entire interface, though, a delegate merely specifies the form of a single function. Also, interfaces are created at compile time and are a fixed aspect of a type, whereas delegates are created at runtime and can be used to dynamically hook up callbacks between objects that weren't originally designed to work together.

Delegates are used as the basis for events in C#, which are the general-purpose notification mechanisms used by the .NET Framework.

Using Delegates

The specification of the delegate determines the form of the function, and to create an instance of the delegate, you must use a function that matches that form. Delegates are sometimes referred to as *safe function pointers*, which isn't a bad analogy, but they do a lot more than act as function pointers.

Because of their dynamic nature, delegates are useful when the user may want to change behavior. If, for example, a collection class implements sorting, it might want to support different sort orders. You could control the sorting based on a delegate that defines the comparison function:

```
using System;
public class Container
{
    public delegate int CompareItemsCallback(object obj1, object obj2);
    public void Sort(CompareItemsCallback compare)
    {
        // not a real sort, just shows what the
        // inner loop code might do
        int x = 0;
        int y = 1;
        object    item1 = arr[x];
        object item2 = arr[y];
        int order = compare(item1, item2);
    }
```

```
    object[]    arr = new object[1];    // items in the collection
}
public class Employee
{
    Employee(string name, int id)
    {
        this.name = name;
        this.id = id;
    }
    public static int CompareName(object obj1, object obj2)
    {
        Employee emp1 = (Employee) obj1;
        Employee emp2 = (Employee) obj2;
        return(String.Compare(emp1.name, emp2.name));
    }
    public static int CompareId(object obj1, object obj2)
    {
        Employee emp1 = (Employee) obj1;
        Employee emp2 = (Employee) obj2;

        if (emp1.id > emp2.id)
            return(1);
        if (emp1.id < emp2.id)
            return(-1);
        else
            return(0);
    }
    string    name;
    int    id;
}
class Test
{
    public static void Main()
    {
        Container employees = new Container();
        // create and add some employees here

            // create delegate to sort on names, and do the sort
        Container.CompareItemsCallback sortByName =
            new Container.CompareItemsCallback(Employee.CompareName);
        employees.Sort(sortByName);
            // employees is now sorted by name
    }
}
```

The delegate defined in the `Container` class takes the two objects to be compared as parameters and returns an integer that specifies the ordering of the two objects. Two static functions are declared that match this delegate as part of the `Employee` class, with each function describing a different kind of ordering.

When the container needs to be sorted, you can pass in a delegate that describes the ordering that should be used, and the sort function will do the sorting. (Well, it would if it were actually implemented.)

Delegates to Instance Members

Users who are familiar with C++ will find a lot of similarity between delegates and C++ function pointers, but there's more to a delegate than there is to a function pointer.

When dealing with Windows functions, it's fairly common to pass in a function pointer that should be called when a specific event occurs. Since C++ function pointers can refer only to static functions and not member functions,[1] you need some way to communicate some state information to the function so it knows to which object the event corresponds. Most functions deal with this by taking a pointer, which is passed through to the callback function. The parameter (in C++ at least) is then cast to a class instance, and then the event is processed.

In C#, delegates can encapsulate both a function to call and an instance to call it on, so you don't need an extra parameter to carry the instance information. This is also a type-safe mechanism, because the instance is specified at the same time the function to call is specified, like so:

```
using System;
public class User
{
    string name;
    public User(string name)
    {
        this.name = name;
    }
    public void Process(string message)
    {
        Console.WriteLine("{0}: {1}", name, message);
    }
}
class Test
{
    delegate void ProcessHandler(string message);
```

1. You might ask, what about member function pointers? Member functions indeed do something similar, but the syntax is rather opaque. Most agree that delegates in C# are both easier to use and more functional.

```
    public static void Main()
    {
        User aUser = new User("George");
        ProcessHandler ph = new ProcessHandler(aUser.Process);

        ph("Wake Up!");
    }
}
```

In this example, a delegate is created that points to the `User.Process()` function, with the `aUser` instance, and the call through the delegate is identical to calling `aUser.Process()` directly.

Multicasting

As mentioned earlier, a delegate can refer to more than one function. Basically, a delegate encapsulates a list of functions that should be called in order. The `Delegate` class provides functions; they take two delegates and return one that encapsulates both or that removes a delegate from another.

To combine two delegates, you can use the `Delegate.Combine()` function. For example, you can easily modify the previous example to call more than one function:

```
using System;
public class User
{
    string name;
    public User(string name)
    {
        this.name = name;
    }
    public void Process(string message)
    {
        Console.WriteLine("{0}: {1}", name, message);
    }
}

class Test
{
    delegate void ProcessHandler(string message);

    static public void Process(string message)
    {
        Console.WriteLine("Test.Process(\"{0}\")", message);
    }
```

```
    public static void Main()
    {
        User user = new User("George");

        ProcessHandler ph = new ProcessHandler(user.Process);
        ph = (ProcessHandler) Delegate.Combine(ph, new ProcessHandler(Process));

        ph("Wake Up!");
    }
}
```

Invoking `ph` now calls both delegates.

This approach has a couple of problems, however. The first is that it's not the best form. More important, however, is that it isn't type-safe at compile time; `Delegate.Combine()` both takes and returns the type `Delegate`, so you have no way at compile time to know whether the delegates are compatible.

To address these issues, C# allows the `+=` and `-=` operators to be used to call `Delegate.Combine()` and `Delegate.Remove()`, and it makes sure the types are compatible. You can modify the call in the example like so:

```
ph += new ProcessHandler(Process);
```

When invoked, the subdelegates encapsulated in a delegate are called synchronously in the order they were added to the delegate. If an exception is thrown by one of the subdelegates, the remaining subdelegates won't be called. If this behavior isn't desirable, you can obtain the list of subdelegates (otherwise known as an *invocation list*) from the delegate and call each subdelegate directly. Instead of this:

```
ph("Wake Up!");
```

you can use the following:

```
foreach (ProcessHandler phDel in ph.GetInvocationList())
{
    try
    {
        phDel("Wake Up!");
    }
    catch (Exception e)
    {
        // log the exception here...
    }
}
```

You can also use this code to implement "blackball" voting, where you can call all delegates once to see if they're able to perform a function and then call them a second time if they all voted yes.

Wanting to call more than one function may seem to be a rare situation, but it's common when dealing with events, which are covered in Chapter 24.

Delegates As Static Members

One drawback of this approach is that the user who wants to use the sorting has to create an instance of the delegate with the appropriate function. It'd be nicer if they didn't have to do that. This can be avoided by defining the appropriate delegates as static members of `Employee`:

```
using System;
public class Container
{
    public delegate int CompareItemsCallback(object obj1, object obj2);
    public void Sort(CompareItemsCallback compare)
    {
        // not a real sort, just shows what the
        // inner loop code might do
        int x = 0;
        int y = 1;
        object    item1 = arr[x];
        object item2 = arr[y];
        int order = compare(item1, item2);
    }
    object[]    arr = new object[1];    // items in the collection
}
class Employee
{
    Employee(string name, int id)
    {
        this.name = name;
        this.id = id;
    }
    public static readonly Container.CompareItemsCallback SortByName =
        new Container.CompareItemsCallback(CompareName);
    public static readonly Container.CompareItemsCallback SortById =
        new Container.CompareItemsCallback(CompareId);

    public static int CompareName(object obj1, object obj2)
    {
        Employee emp1 = (Employee) obj1;
        Employee emp2 = (Employee) obj2;
        return(String.Compare(emp1.name, emp2.name));
    }
    public static int CompareId(object obj1, object obj2)
    {
        Employee emp1 = (Employee) obj1;
        Employee emp2 = (Employee) obj2;
```

```
        if (emp1.id > emp2.id)
            return(1);
        if (emp1.id < emp2.id)
            return(-1);
        else
            return(0);
    }
    string    name;
    int    id;
}
class Test
{
    public static void Main()
    {
        Container employees = new Container();
        // create and add some employees here

        employees.Sort(Employee.SortByName);
            // employees is now sorted by name
    }
}
```

This is a lot easier. The users of `Employee` don't have to know how to create the delegate—they can just refer to the static member.

Delegates As Static Properties

One thing that's a bit wasteful, however, is that the delegate is always created, even if it's never used. It'd be better if the delegate were created on the fly as needed. You can do this by replacing the static members with properties:

```
using System;
class Container
{
    public delegate int CompareItemsCallback(object obj1, object obj2);
    public void SortItems(CompareItemsCallback compare)
    {
        // not a real sort, just shows what the
        // inner loop code might do…
        int x = 0;
        int y = 1;
        object    item1 = arr[x];
        object item2 = arr[y];
        int order = compare(item1, item2);
    }
    object[]    arr;    // items in the collection
}
```

```
class Employee
{
    Employee(string name, int id)
    {
        this.name = name;
        this.id = id;
    }
    public static Container.CompareItemsCallback SortByName
    {
        get
        {
            return(new Container.CompareItemsCallback(CompareName));
        }
    }
    public static Container.CompareItemsCallback SortById
    {
        get
        {
            return(new Container.CompareItemsCallback(CompareId));
        }
    }
    static int CompareName(object obj1, object obj2)
    {
        Employee emp1 = (Employee) obj1;
        Employee emp2 = (Employee) obj2;
        return(String.Compare(emp1.name, emp2.name));
    }
    static int CompareId(object obj1, object obj2)
    {
        Employee emp1 = (Employee) obj1;
        Employee emp2 = (Employee) obj2;

        if (emp1.id > emp2.id)
            return(1);
        if (emp1.id < emp2.id)
            return(-1);
        else
            return(0);
    }
    string   name;
    int    id;
}
class Test
{
```

```
    public static void Main()
    {
        Container employees = new Container();
        // create and add some employees here

        employees.SortItems(Employee.SortByName);
            // employees is now sorted by name
    }
}
```

With this version, rather than `Employee.SortByName` being a delegate, it's a function that returns a delegate that can sort by name.

Initially, this example had the private static delegate members `SortByName` and `SortById`, and the property created the static member if it hadn't been used before. This works well if creating the delegate is somewhat costly and the delegate is likely to be used again.

In this case, however, it's much easier to create the delegate on the fly and just return it to the user. As soon as the `Sort` function on `Container` is done with the delegate, it will be available for collection by the garbage collector.

■ **Note** This example is for illustration only. To implement a collection class that does sorting, the techniques used by the framework are much easier. See Chapter 29 for more information.

Anonymous Methods

At various times, the code contained within the body of a delegate will be quite small and used in only one place. Many developers prefer to forward all event handlers to another method so that other event handlers can reuse the same code logic. In situations such as these, creating a method just so the call can be forwarded to another method isn't exactly elegant, and it's preferable to have the ability to include the code that forwards the call to the end recipient inline with the delegate creation. C# 2.0 provides this functionality through a new feature called *anonymous methods*.

Anonymous methods offload the tedious work of declaring a separate method to the compiler. Consider the following code, which is fairly verbose:

```
class Program
{
 static void Main(string[] args)
 {
  AppDomain.CurrentDomain.CatchingException +=
  new CatchingExceptionEventHandler(CurrentDomain_CatchingException);
 }

 static void CurrentDomain_CatchingException(object sender,
```

```
  CatchingExceptionEventArgs e)
 {
  Console.WriteLine("An exception is being caught in the app domain");
 }
}
```

Using anonymous methods, you can convert this code to the following:

```
class Program
{
 static void Main(string[] args)
 {
  AppDomain.CurrentDomain.CatchingException +=
  delegate(object sender, CatchingExceptionEventArgs e)
  {
   Console.WriteLine("An exception is being caught in the app domain");
  };
 }
}
```

If the code inside the anonymous method doesn't need to access the values of the delegate's parameters, you can exclude these to further simplify the code:

```
class Program
{
 static void Main(string[] args)
 {
  AppDomain.CurrentDomain.CatchingException +=
  delegate  {
   Console.WriteLine("An exception is being caught in the app domain");
  };
 }
}
```

In the anonymous method snippets, the event hookup and delegate declaration take place in the same line. Although this isn't a profound language improvement, it can be quite convenient and gives the code a more natural flow when the delegate body is short.

Anonymous methods are extracted by the compiler and placed inside a generated method with parameters appropriate to the delegate. An additional benefit of anonymous methods is that the generated method won't appear in Visual Studio's IntelliSense feature, and the method name contains characters that aren't legal to use in C# (although it's legal from the CLR's perspective), so it can't be called directly in C#. You shouldn't rely upon the inability to call the method in C# as a security measure, and it's still possible to call the method using reflection (more about this in Chapter 38) or using a different language.

Anonymous methods don't expose the full power of delegates. Because no delegate instance is explicitly created in the code, it isn't possible to unsubscribe to event handlers, because this requires a reference to the delegate instance for which the subscription should be removed. In most circumstances, this isn't overly limiting, because the lifetime of the raiser and listener will be identical, but if the object raising the events is long-lived, the inability to unsubscribe will

result in the event raiser's reference preventing the collection of event listeners that are no longer referenced by any other object.

If a delegate defines `ref` or `out` parameters, anonymous methods can't be used in place of the delegate, as anonymous methods don't support the ability to deal with these parameters.

One of the more interesting features of anonymous method is the ability to access the local variables of the method that declares the anonymous method. To achieve the same outcome without anonymous methods, the local variables need to be elevated to class member variables so they can be accessed by the delegate instantiator and the delegate method. This is similar to what the compiler does when the delegate instantiator's local variables (which are termed *outer variables*) are accessed inside the anonymous method. Consider the following example, where the generic `ForEach` method of `Array` calculates the sum on each element in an array:

```
int sum = 0;
int[] arr = new int[] { 1, 2, 3 };
Array.ForEach(arr, delegate(int i) { sum += i; });
Console.WriteLine(sum);
```

In this case, the outer variable is read from and written to in the anonymous method. In this case, the compiler will generate a private nested class with a name containing characters that will prevent C# code from accessing it directly. The compiler will also use the nested class to store the value of the `sum` outer variable.

CHAPTER 24

Events

Chapter 23 showed how to use delegates to pass a reference to a method so it can be called in a general way. Being able to call a method in such a way is useful in graphical user interfaces, such as the one provided by the classes in `System.Windows.Forms`. It's fairly easy to build such a framework by using delegates, as shown in this example:

```
using System;

public class Button
{
    public delegate void ClickHandler(object sender, EventArgs e);
    public ClickHandler Click;

    protected void OnClick()
    {
        if (Click != null)
            Click(this, null);

    }

    public void SimulateClick()
    {
        OnClick();
    }
}

class Test
{
    static public void ButtonHandler(object sender, EventArgs e)
    {
        Console.WriteLine("Button clicked");
    }
```

```
    public static void Main()
    {
        Button button = new Button();

        button.Click = new Button.ClickHandler(ButtonHandler);

        button.SimulateClick();
    }
}
```

The `Button` class is supporting a click "event"[1] by having the `ClickHandler` delegate tell what kind of method can be called to hook up, and a delegate instance can then be assigned to the event. The `OnClick()` method then calls this delegate, and everything works fine—at least in this simple case.

The situation gets more complicated in a real-world scenario. In real applications, a button such as this one lives in a form, and clicking the button might be of interest to more than one area of the application. Doing this isn't a problem with delegates because more than one method can be called from a single delegate instance. In the previous example, if another class also wanted to be called when the button was clicked, the += operator can be used, like this:

```
button.Click += new Button.ClickHandler(OtherMethodToCall);
```

Unfortunately, if the other class wasn't careful, it might do the following:

```
button.Click = new Button.ClickHandler(OtherMethodToCall);
```

This would be bad, as it'd mean that your `ButtonHandler` would be unhooked and only the new method would be called.

Similarly, to unhook from the click, the right thing to do is use this code:[2]

```
button.Click -= new Button.ClickHandler(OtherMethodToCall);
```

but the following might be used instead:

```
button.Click = null;
```

This is also wrong.

What you need is some way of protecting the delegate field so it's accessed only using += and -=.

Add and Remove Functions

An easy way to do this is to make the delegate field private and write a couple of methods that can be used to add or remove delegates:

1. This isn't an "event" in the C# sense of the word but just the abstract concept of something happening.
2. This syntax may look weird because a new instance of the delegate is created just so it can be removed from the delegate. When `Delegate.Remove()` is called, it needs to find the delegate in the invocation list, so a delegate instance is required.

```
using System;

public class Button
{
    public delegate void ClickHandler(object sender, EventArgs e);
    private ClickHandler click;

    public void AddClick(ClickHandler clickHandler)
    {
        click += clickHandler;
    }

    public void RemoveClick(ClickHandler clickHandler)
    {
        click -= clickHandler;
    }

    protected void OnClick()
    {
        if (click != null)
            click(this, null);

    }

    public void SimulateClick()
    {
        OnClick();
    }
}

class Test
{
    static public void ButtonHandler(object sender, EventArgs e)
    {
        Console.WriteLine("Button clicked");
    }

    public static void Main()
    {
        Button button = new Button();

        button.AddClick(new Button.ClickHandler(ButtonHandler));

        button.SimulateClick();

        button.RemoveClick(new Button.ClickHandler(ButtonHandler));
    }
}
```

In this example, the AddClick() and RemoveClick() methods have been added, and the delegate field is now private. It's now impossible for users of the class to do the wrong thing when they hook or unhook.

This example is reminiscent of the example in Chapter 19. It had two accessor methods, and adding properties made those two methods look like a field. Let's add a feature to the compiler so there's a "virtual" delegate named Click. It can write the AddClick() and RemoveClick() methods for you, and it can also change a use of += or -= to the appropriate add or remove call. This gives you the advantage of having the Add and Remove methods without having to write them.

You need a keyword for this compiler enhancement, and event seems like a good choice:

```
using System;

public class Button
{
    public delegate void ClickHandler(object sender, EventArgs e);

    public event ClickHandler Click;

    protected void OnClick()
    {
        if (Click != null)
            Click(this, null);
    }

    public void SimulateClick()
    {
        OnClick();
    }
}

class Test
{
    static public void ButtonHandler(object sender, EventArgs e)
    {
        Console.WriteLine("Button clicked");
    }

    public static void Main()
    {
        Button button = new Button();

        button.Click += new Button.ClickHandler(ButtonHandler);

        button.SimulateClick();

        button.Click -= new Button.ClickHandler(ButtonHandler);
    }
}
```

When the event keyword is added to a delegate, the compiler creates a private field and then writes public add_Click() and remove_Click() methods. It also emits a bit of metadata that says there's an event named Click and that event is associated with add and remove methods with these names so that object browsers and such can tell there's an event on this class.

In Main(), the event is accessed like a delegate; but since the add and remove methods are the only ways to access the private delegate, += and -= are the only operations that can be performed on the event.

That's the basic story for events. The arguments to the event handler, object sender and EventArgs e, are by convention and should be followed by other classes that expose events. The sender argument allows the user of the code to know which object fired the event, and the e argument contains the information associated with the event. In this case, there's no additional information to pass, so EventArgs is used. If additional information needed to be passed, a class should be derived from EventArgs with the additional information. For example, the KeyEventArgs class in the framework looks like this:

```
using System;
using System.Windows.Forms;
class KeyEventArgs: EventArgs
{
    Keys    keyData;

    KeyEventArgs(Keys keyData)
    {
        this.keyData = keyData;
    }

    public Keys KeyData
    {
        get
        {
            return(keyData);
        }
    }

    // other functions here...
}
```

The OnKey method will take a parameter of type Keys, encapsulate it into a KeyEventArgs class, and then call the delegate.

Custom Add and Remove

Because the compiler creates a private delegate field for every event that's declared, a class that declares numerous events will use one field per event. The Control class in System.Windows.Forms declares more than 25 events, but there are usually just a couple of these events hooked up for a given control. What's needed is a way to avoid allocating the storage for the delegate unless it's needed.

The C# language supports this by allowing the `add()` and `remove()` methods to be written directly, which lets delegates be stored in a more space-efficient manner. One typical way of doing this is to define a `Hashtable` as part of the object and then to store the delegate in the `Hashtable`, like this:

```
using System;
using System.Collections;
using System.Runtime.CompilerServices;
public class Button
{
    public delegate void ClickHandler(object sender, EventArgs e);

    Hashtable delegateStore = new Hashtable();
    static object clickEventKey = new object();

    public event ClickHandler Click
    {
        [MethodImpl(MethodImplOptions.Synchronized)]
        add
        {
            delegateStore[clickEventKey] =
                Delegate.Combine((Delegate) delegateStore[clickEventKey],
                         value);
        }

        [MethodImpl(MethodImplOptions.Synchronized)]
        remove
        {
            delegateStore[clickEventKey] =
                Delegate.Remove((Delegate) delegateStore[clickEventKey],
                         value);
        }
    }

    protected void OnClick()
    {
        ClickHandler ch = (ClickHandler) delegateStore[clickEventKey];
        if (ch != null)
            ch(this, null);
    }

    public void SimulateClick()
    {
        OnClick();
    }
}
```

```
class Test
{
    static public void ButtonHandler(object sender, EventArgs e)
    {
        Console.WriteLine("Button clicked");
    }

    public static void Main()
    {
        Button button = new Button();

        button.Click += new Button.ClickHandler(ButtonHandler);

        button.SimulateClick();

        button.Click -= new Button.ClickHandler(ButtonHandler);
    }
}
```

The `add()` and `remove()` methods are written using a syntax similar to the one used for properties, and they use the `delegateStore` hash table to store the delegate. One problem with using a hash table is coming up with a key that can be used to store and fetch the delegates. There's nothing associated with an event that can serve as a unique key, so `clickEventKey` is an object that's included only so you can use it as a key for the hash table. It's static because the same unique value can be used for all instances of the `Button` class.

The `MethodImpl` attribute is required so two threads won't try to add or remove delegates at the same time (which would be bad). You could also do this with the `lock` statement.

This implementation still results in the use of one field per object for the hash table. To get rid of this, you have a couple of options. The first is to make the hash table static so that it can be shared among all instances of the `Button` class. The second choice is to make a single global class to be shared among all the controls, which saves the most space.[3]

Both are good choices, but both will require a couple of changes to the approach. First, simply using an object as a key isn't good enough since the hash table is shared among all the instances of the object, so the instance will also have to be used as a key.

A subtle outgrowth of using the instance is that the controls have to call a method to remove their event storage when they're closed. If this didn't happen, the control wouldn't be visible anymore, but it'd still be referenced through the global event object, and the memory would never be reclaimed by the garbage collector.

Here's the a final version of the example, with a global delegate cache:

3. But it perhaps provides slower access, since this hash table will have more entries than a per-control implementation.

```
using System;
using System.Collections;
using System.Runtime.CompilerServices;

    //
    // Global delegate cache. Uses a two-level hash table. The delegateStore
    // hash table stores a hash table keyed on the object instance, and the
    // instance hash table is keyed on the unique key. This allows fast teardown
    // of the object when it's destroyed.
    //
public class DelegateCache
{
    private DelegateCache() {}    // nobody can create one of these

    Hashtable delegateStore = new Hashtable();    // top level hash table

    static DelegateCache dc = new DelegateCache();    // our single instance

    Hashtable GetInstanceHash(object instance)
    {
        Hashtable instanceHash = (Hashtable) delegateStore[instance];

        if (instanceHash == null)
        {
            instanceHash = new Hashtable();
            delegateStore[instance] = instanceHash;

        }
        return(instanceHash);
    }

    public static void Combine(Delegate myDelegate, object instance, object key)
    {
        lock(instance)
        {
            Hashtable instanceHash = dc.GetInstanceHash(instance);

            instanceHash[key] =
                Delegate.Combine((Delegate) instanceHash[key],  myDelegate);
        }
    }

    public static void Remove(Delegate myDelegate, object instance, object key)
    {
        lock(instance)
        {
            Hashtable instanceHash = dc.GetInstanceHash(instance);
```

```
            instanceHash[key] =
                Delegate.Remove((Delegate) instanceHash[key], myDelegate);
        }
    }

    public static Delegate Fetch(object instance, object key)
    {
        Hashtable instanceHash = dc.GetInstanceHash(instance);

        return((Delegate) instanceHash[key]);
    }

    public static void ClearDelegates(object instance)
    {
        dc.delegateStore.Remove(instance);
    }
}

public class Button
{
    public void TearDown()
    {
        DelegateCache.ClearDelegates(this);
    }


    public delegate void ClickHandler(object sender, EventArgs e);

    static object clickEventKey = new object();

    public event ClickHandler Click
    {
        add
        {
            DelegateCache.Combine(value, this, clickEventKey);
        }

        remove
        {
            DelegateCache.Remove(value, this, clickEventKey);
        }
    }
```

```
    protected void OnClick()
    {
        ClickHandler ch = (ClickHandler) DelegateCache.Fetch(this, clickEventKey);

        if (ch != null)
            ch(this, null);
    }

    public void SimulateClick()
    {
        OnClick();
    }
}

class Test
{
    static public void ButtonHandler(object sender, EventArgs e)
    {
        Console.WriteLine("Button clicked");
    }

    public static void Main()
    {
        Button button = new Button();

        button.Click += new Button.ClickHandler(ButtonHandler);

        button.SimulateClick();

        button.Click -= new Button.ClickHandler(ButtonHandler);

        button.TearDown();
    }
}
```

The `DelegateCache` class stores the hash tables for each instance that has stored a delegate in a main hash table. This allows for easier cleanup when a control is finished. The `Combine()`, `Remove()`, and `Fetch()` methods do what's expected. The `ClearDelegates()` method is called by `Button.TearDown()` to remove all delegates stored for a specific control instance.[4]

4. In a real implementation, this could be done through a teardown function, or it could also be done through a `Dispose()` method.

CHAPTER 25

User-Defined Conversions

C# allows conversions to be defined between classes or structs and other objects in the system. User-defined conversions are always static functions, which must either take as a parameter or return as a return value the object in which they're declared. This means conversions can't be declared between two existing types, which makes the language simpler.

A Simple Example

The following example implements a struct that handles roman numerals. You could also write it as a class, but since it's a piece of data, a struct makes more sense.

```
using System;
using System.Text;
struct RomanNumeral
{
    public RomanNumeral(short value)
    {
        if (value > 5000)
            throw(new ArgumentOutOfRangeException());

        this.value = value;
    }
    public static explicit operator RomanNumeral(
        short value)
    {
        RomanNumeral    retval;
        retval = new RomanNumeral(value);
        return(retval);
    }

    public static implicit operator short(
    RomanNumeral roman)
    {
        return(roman.value);
    }
```

```
    static string NumberString(
    ref int value, int magnitude, char letter)
    {
        StringBuilder    numberString = new StringBuilder();

        while (value >= magnitude)
        {
            value -= magnitude;
            numberString.Append(letter);
        }
        return(numberString.ToString());
    }

    public static implicit operator string(
    RomanNumeral roman)
    {
        int        temp = roman.value;

        StringBuilder retval = new StringBuilder();

        retval.Append(RomanNumeral.NumberString(ref temp, 1000, 'M'));
        retval.Append(RomanNumeral.NumberString(ref temp, 500, 'D'));
        retval.Append(RomanNumeral.NumberString(ref temp, 100, 'C'));
        retval.Append(RomanNumeral.NumberString(ref temp, 50, 'L'));
        retval.Append(RomanNumeral.NumberString(ref temp, 10, 'X'));
        retval.Append(RomanNumeral.NumberString(ref temp, 5, 'V'));
        retval.Append(RomanNumeral.NumberString(ref temp, 1, 'I'));

        return(retval.ToString());
    }

    private short value;
}
class Test
{
    public static void Main()
    {
        short s = 12;
        RomanNumeral numeral = new RomanNumeral(s);

        s = 165;
        numeral = (RomanNumeral) s;
```

```
        Console.WriteLine("Roman as int: {0}", (int)numeral);
        Console.WriteLine("Roman as string: {0}", (string)numeral);

        short s2 = numeral;
    }
}
```

This struct declares a constructor that can take a `short` value, and it also declares a conversion from an integer to a `RomanNumeral`. The conversion is declared as an `explicit` conversion because it may throw an exception if the number is bigger than the magnitude supported by the struct. You'll see a conversion to `short` that's declared `implicit`, because the value in a `RomanNumeral` will always fit in a `short`. And finally, you'll see a conversion to `string` that gives the romanized version of the number.[1]

When you create an instance of this struct, you can use the constructor to set the value. You can use an explicit conversion to convert the integer value to a `RomanNumeral`. To get the romanized version of the `RomanNumeral`, write the following:

```
Console.WriteLine(roman);
```

If you do this, the compiler reports that an ambiguous conversion is present. The class includes `implicit` conversions both to `short` and to `string`, and `Console.WriteLine()` has overloads that take both versions, so the compiler doesn't know which one to call.

The example uses an `explicit` cast to disambiguate, but it's a bit ugly. Since this struct will likely be used primarily to print the romanized notation, it probably makes sense to change the conversion to the integer to be an `explicit` one so that the conversion to `string` is the only `implicit` one.

Pre- and Post-Conversions

In the preceding example, the basic types that were converted to and from the `RomanNumeral` were exact matches to the types declared in the struct itself. You can also use the user-defined conversions when the source or destination types aren't exact matches to the types in the conversion functions.

If the source or destination types aren't exact matches, then the appropriate standard (in other words, built-in) conversion must be present to convert from the source type to the source type of the user-defined conversion and/or from the destination type of the user-defined conversion, and the type of the conversion (`implicit` or `explicit`) must also be compatible.

Perhaps an example will be a bit easier to understand. The following line calls the implicit user-defined conversion directly:

```
short s = numeral;
```

Since this is an implicit use of the user-defined conversion, another implicit conversion can appear at the end:

```
int i = numeral;
```

1. This struct doesn't handle niceties such as replacing "IIII" with "IV"; it also doesn't handle converting the romanized string to a `short`. We'll leave the remainder of the implementation as an exercise for the reader.

Here, the implicit conversion from `RomanNumeral` to `short` is performed, followed by the implicit conversion from `short` to `long`.

In the explicit case, the example has the following conversion:

```
numeral = (RomanNumeral) 165;
```

Since the usage is `explicit`, the explicit conversion from `int` to `RomanNumeral` is used. Also, an explicit conversion can occur before the user-defined conversion is called:

```
long bigvalue = 166;
short smallvalue = 12;
numeral = (RomanNumeral) bigvalue;
numeral = (RomanNumeral) smallvalue;
```

In the first conversion, the `long` value is converted by explicit conversion to an integer, and then the user-defined conversion is called. The second conversion is similar, except that an implicit conversion is performed before the explicit user-defined conversion.

Conversions Between Structs

User-defined conversions that deal with classes or structs rather than basic types work similarly, except you have a few more situations to consider. Since you can define the user conversion in either the source type or the destination type, you have a bit more design work to do, and the operation is a bit more complex. For details, see the "How It Works" section.

Building on the `RomanNumeral` example in the previous section, you can add a struct that handles binary numbers like so:

```
using System;
using System.Text;
struct RomanNumeral
{
    public RomanNumeral(short value)
    {
        if (value > 5000)
            throw(new ArgumentOutOfRangeException());

        this.value = value;
    }
    public static explicit operator RomanNumeral(
    short value)
    {
        RomanNumeral    retval;
        retval = new RomanNumeral(value);
        return(retval);
    }
```

```
    public static implicit operator short(
    RomanNumeral roman)
    {
        return(roman.value);
    }

    static string NumberString(
    ref int value, int magnitude, char letter)
    {
        StringBuilder    numberString = new StringBuilder();

        while (value >= magnitude)
        {
            value -= magnitude;
            numberString.Append(letter);
        }
        return(numberString.ToString());
    }

    public static implicit operator string(
    RomanNumeral roman)
    {
        int        temp = roman.value;

        StringBuilder retval = new StringBuilder();

        retval.Append(RomanNumeral.NumberString(ref temp, 1000, 'M'));
        retval.Append(RomanNumeral.NumberString(ref temp, 500, 'D'));
        retval.Append(RomanNumeral.NumberString(ref temp, 100, 'C'));
        retval.Append(RomanNumeral.NumberString(ref temp, 50, 'L'));
        retval.Append(RomanNumeral.NumberString(ref temp, 10, 'X'));
        retval.Append(RomanNumeral.NumberString(ref temp, 5, 'V'));
        retval.Append(RomanNumeral.NumberString(ref temp, 1, 'I'));

        return(retval.ToString());
    }

    private short value;
}
struct BinaryNumeral
{
    public BinaryNumeral(int value)
    {
        this.value = value;
    }
```

```
    public static implicit operator BinaryNumeral(
    int value)
    {
        BinaryNumeral    retval = new BinaryNumeral(value);
        return(retval);
    }

    public static implicit operator int(
    BinaryNumeral binary)
    {
        return(binary.value);
    }

    public static implicit operator string(
    BinaryNumeral binary)
    {
        StringBuilder    retval = new StringBuilder();

        return(retval.ToString());
    }

    private int value;
}
class Test
{
    public static void Main()
    {
        RomanNumeral    roman = new RomanNumeral(12);
        BinaryNumeral    binary;
        binary = (BinaryNumeral)(int)roman;
    }
}
```

You can use the classes together, but since they don't really know about each other, it takes a bit of extra typing. Converting from a `RomanNumeral` to a `BinaryNumeral` requires first converting to an `int`.

It'd be nice to write the `Main()` function as follows and make the types look like the built-in types, with the exception that `RomanNumeral` has a smaller range than binary and therefore will require an explicit conversion in that section:

```
binary = roman;
roman = (RomanNumeral) binary;
```

To get this, a user-defined conversion is required on either the `RomanNumeral` class or the `BinaryNumeral` class. In this case, it goes on the `RomanNumeral` class (for reasons that should become clear in the "Design Guidelines" section of this chapter).

You can modify the classes as follows, adding two conversions:

```
using System;
using System.Text;
struct RomanNumeral
{
    public RomanNumeral(short value)
    {
        if (value > 5000)
            throw(new ArgumentOutOfRangeException());

        this.value = value;
    }
    public static explicit operator RomanNumeral(
    short value)
    {
        RomanNumeral    retval;
        retval = new RomanNumeral(value);
        return(retval);
    }

    public static implicit operator short(
    RomanNumeral roman)
    {
        return(roman.value);
    }

    static string NumberString(
    ref int value, int magnitude, char letter)
    {
        StringBuilder    numberString = new StringBuilder();

        while (value >= magnitude)
        {
            value -= magnitude;
            numberString.Append(letter);
        }
        return(numberString.ToString());
    }

    public static implicit operator string(
    RomanNumeral roman)
    {
        int        temp = roman.value;

        StringBuilder retval = new StringBuilder();
```

```
            retval.Append(RomanNumeral.NumberString(ref temp, 1000, 'M'));
            retval.Append(RomanNumeral.NumberString(ref temp, 500, 'D'));
            retval.Append(RomanNumeral.NumberString(ref temp, 100, 'C'));
            retval.Append(RomanNumeral.NumberString(ref temp, 50, 'L'));
            retval.Append(RomanNumeral.NumberString(ref temp, 10, 'X'));
            retval.Append(RomanNumeral.NumberString(ref temp, 5, 'V'));
            retval.Append(RomanNumeral.NumberString(ref temp, 1, 'I'));

            return(retval.ToString());
        }
        public static implicit operator BinaryNumeral(RomanNumeral roman)
        {
            return(new BinaryNumeral((short) roman));
        }

        public static explicit operator RomanNumeral(
        BinaryNumeral binary)
        {
            return(new RomanNumeral((short) binary));
        }

        private short value;
    }
    struct BinaryNumeral
    {
        public BinaryNumeral(int value)
        {
            this.value = value;
        }
        public static implicit operator BinaryNumeral(
        int value)
        {
            BinaryNumeral    retval = new BinaryNumeral(value);
            return(retval);
        }

        public static implicit operator int(
        BinaryNumeral binary)
        {
            return(binary.value);
        }

        public static implicit operator string(
        BinaryNumeral binary)
        {
            StringBuilder    retval = new StringBuilder();
```

```
        return(retval.ToString());
    }

    private int value;
}
class Test
{
    public static void Main()
    {
        RomanNumeral    roman = new RomanNumeral(122);
        BinaryNumeral    binary;
        binary = roman;
        roman = (RomanNumeral) binary;
    }
}
```

With these added conversions, conversions between the two types can now take place.

Classes and Pre- and Post-Conversions

As with basic types, classes can have standard conversions that occur either before or after the user-defined conversion, or even before *and* after. The only standard conversions that deal with classes, however, are conversions to a base or derived class, so those are the only ones covered in this section.

Implicit conversions are pretty simple; the conversion occurs in three steps:

1. A conversion from a derived class to the source class of the user-defined conversion is optionally performed.
2. The user-defined conversion occurs.
3. A conversion from the destination class of the user-defined conversion to a base class is optionally performed.

To illustrate this, you can modify the example to use classes rather than structs and add a new class that derives from `RomanNumeral`:

```
using System;
using System.Text;
class RomanNumeral
{
    public RomanNumeral(short value)
    {
        if (value > 5000)
            throw(new ArgumentOutOfRangeException());

        this.value = value;
    }
```

```
public static explicit operator RomanNumeral(
short value)
{
    RomanNumeral    retval;
    retval = new RomanNumeral(value);
    return(retval);
}

public static implicit operator short(
RomanNumeral roman)
{
    return(roman.value);
}

static string NumberString(
ref int value, int magnitude, char letter)
{
    StringBuilder    numberString = new StringBuilder();

    while (value >= magnitude)
    {
        value -= magnitude;
        numberString.Append(letter);
    }
    return(numberString.ToString());
}

public static implicit operator string(
RomanNumeral roman)
{
    int        temp = roman.value;

    StringBuilder retval = new StringBuilder();

    retval.Append(RomanNumeral.NumberString(ref temp, 1000, 'M'));
    retval.Append(RomanNumeral.NumberString(ref temp, 500, 'D'));
    retval.Append(RomanNumeral.NumberString(ref temp, 100, 'C'));
    retval.Append(RomanNumeral.NumberString(ref temp, 50, 'L'));
    retval.Append(RomanNumeral.NumberString(ref temp, 10, 'X'));
    retval.Append(RomanNumeral.NumberString(ref temp, 5, 'V'));
    retval.Append(RomanNumeral.NumberString(ref temp, 1, 'I'));

    return(retval.ToString());
}
```

```
    public static implicit operator BinaryNumeral(RomanNumeral roman)
    {
        return(new BinaryNumeral((short) roman));
    }

    public static explicit operator RomanNumeral(
    BinaryNumeral binary)
    {
        return(new RomanNumeral((short)(int) binary));
    }

    private short value;
}
class BinaryNumeral
{
    public BinaryNumeral(int value)
    {
        this.value = value;
    }
    public static implicit operator BinaryNumeral(
    int value)
    {
        BinaryNumeral    retval = new BinaryNumeral(value);
        return(retval);
    }

    public static implicit operator int(
    BinaryNumeral binary)
    {
        return(binary.value);
    }

    public static implicit operator string(
    BinaryNumeral binary)
    {
        StringBuilder    retval = new StringBuilder();

        return(retval.ToString());
    }

    private int value;
}
```

```
class RomanNumeralAlternate : RomanNumeral
{
    public RomanNumeralAlternate(short value): base(value)
    {
    }

    public static implicit operator string(
    RomanNumeralAlternate roman)
    {
        return("NYI");
    }
}
class Test
{
    public static void Main()
    {
            // implicit conversion section
        RomanNumeralAlternate    roman;
        roman = new RomanNumeralAlternate(55);

        BinaryNumeral binary = roman;
            // explicit conversion section
        BinaryNumeral binary2 = new BinaryNumeral(1500);
        RomanNumeralAlternate roman2;

        roman2 = (RomanNumeralAlternate) binary2;
    }
}
```

The operation of the `implicit` conversion to `BinaryNumeral` is as expected; an implicit conversion of `roman` from `RomanNumeralAlternate` to `RomanNumeral` occurs, and then the user-defined conversion from `RomanNumeral` to `BinaryNumeral` takes places.

The `explicit` conversion section may have some people scratching their heads. The user-defined function from `BinaryNumeral` to `RomanNumeral` returns a `RomanNumeral`, and the post-conversion to `RomanNumeralAlternate` can never succeed.

You can rewrite the conversion as follows:

```
using System;
using System.Text;
class RomanNumeral
{
    public RomanNumeral(short value)
    {
        if (value > 5000)
            throw(new ArgumentOutOfRangeException());

        this.value = value;
    }
```

```
public static implicit operator short(
RomanNumeral roman)
{
    return(roman.value);
}

static string NumberString(
ref int value, int magnitude, char letter)
{
    StringBuilder    numberString = new StringBuilder();

    while (value >= magnitude)
    {
        value -= magnitude;
        numberString.Append(letter);
    }
    return(numberString.ToString());
}

public static implicit operator string(
RomanNumeral roman)
{
    int        temp = roman.value;

    StringBuilder retval = new StringBuilder();

    retval.Append(RomanNumeral.NumberString(ref temp, 1000, 'M'));
    retval.Append(RomanNumeral.NumberString(ref temp, 500, 'D'));
    retval.Append(RomanNumeral.NumberString(ref temp, 100, 'C'));
    retval.Append(RomanNumeral.NumberString(ref temp, 50, 'L'));
    retval.Append(RomanNumeral.NumberString(ref temp, 10, 'X'));
    retval.Append(RomanNumeral.NumberString(ref temp, 5, 'V'));
    retval.Append(RomanNumeral.NumberString(ref temp, 1, 'I'));

    return(retval.ToString());
}
public static implicit operator BinaryNumeral(RomanNumeral roman)
{
    return(new BinaryNumeral((short) roman));
}
```

```
    public static explicit operator RomanNumeral(
    BinaryNumeral binary)
    {
        int        val = binary;
        if (val >= 1000)
            return((RomanNumeral)
                    new RomanNumeralAlternate((short) val));
        else
            return(new RomanNumeral((short) val));
    }

    private short value;
}
class BinaryNumeral
{
    public BinaryNumeral(int value)
    {
        this.value = value;
    }
    public static implicit operator BinaryNumeral(
    int value)
    {
        BinaryNumeral    retval = new BinaryNumeral(value);
        return(retval);
    }

    public static implicit operator int(
    BinaryNumeral binary)
    {
        return(binary.value);
    }

    public static implicit operator string(
    BinaryNumeral binary)
    {
        StringBuilder    retval = new StringBuilder();

        return(retval.ToString());
    }

    private int value;
}
class RomanNumeralAlternate : RomanNumeral
{
```

```
    public RomanNumeralAlternate(short value) : base(value)
    {
    }

    public static implicit operator string(
    RomanNumeralAlternate roman)
    {
        return("NYI");
    }
}
class Test
{
    public static void Main()
    {
            // implicit conversion section
        RomanNumeralAlternate    roman;
        roman = new RomanNumeralAlternate(55);
        BinaryNumeral binary = roman;

        // explicit conversion section
        BinaryNumeral binary2 = new BinaryNumeral(1500);
        RomanNumeralAlternate roman2;

        roman2 = (RomanNumeralAlternate) binary2;
    }
}
```

The user-defined conversion operator now doesn't return a `RomanNumeral`; it returns a `RomanNumeral` reference to an object, and it's perfectly legal for that to be a reference to a derived type. This is weird, perhaps, but legal. With the revised version of the conversion function, the `explicit` conversion from `BinaryNumeral` to `RomanNumeralAlternate` may succeed, depending on whether the `RomanNumeral` reference is a reference to a `RomanNumeral` object or a `RomanNumeralAlternate` object.

Design Guidelines

When designing user-defined conversions, you should consider the following guidelines.

Implicit Conversions Are Safe Conversions

When defining conversions between types, the only conversions that should be implicit ones are those that don't lose any data and don't throw exceptions.

This is important, because implicit conversions can occur without it being obvious that a conversion has occurred.

Define the Conversion in the More Complex Type

This basically means not cluttering up a simple type with conversions to a more complex one. For conversions to and from one of the predefined types, you have no option but to define the conversion as part of the class, since the source isn't available.

Even if the source is available, however, it's strange to define the conversions from `int` to `BinaryNumeral` or `RomanNumeral` in the `int` class.

Sometimes, as in the example, the classes are peers to each other, and no obvious simpler class exists. In that case, pick a class, and put both conversions there.

One Conversion to and from a Hierarchy

The examples in this chapter had only a single conversion from the user-defined type to the numeric types and one conversion from numeric types to the user-defined type. In general, it's good practice to do this and then use the built-in conversions to move between the destination types. When choosing the numeric type to convert from or to, choose the one that's the most natural size for the type.

For example, the `BinaryNumeral` class contains an implicit conversion to `int`. If the user wants a smaller type, such as `short`, you can easily perform a cast.

If multiple conversions are available, the overloading rules will take effect, and the result may not always be intuitive for the user of the class. This is especially important when dealing with both signed and unsigned types.

Add Conversions Only As Needed

Extraneous conversions only make the user's life harder.

Conversions That Operate in Other Languages

Some of the .NET languages don't support the conversion syntax, and calling conversion functions—which have weird names—may be difficult or impossible. To make classes easily usable from these languages, you should supply alternate versions of the conversions. If, for example, an object supports a conversion to a string, it should also support calling `ToString()` on that function. Here's how you'd do it on the `RomanNumeral` class:

```
using System;
using System.Text;

class RomanNumeral
{
    public RomanNumeral(short value)
    {
        if (value > 5000)
            throw(new ArgumentOutOfRangeException());

        this.value = value;
    }
```

```
public static explicit operator RomanNumeral(
short value)
{
    RomanNumeral    retval;
    retval = new RomanNumeral(value);
    return(retval);
}

public static implicit operator short(
RomanNumeral roman)
{
    return(roman.value);
}

static string NumberString(
ref int value, int magnitude, char letter)
{
    StringBuilder    numberString = new StringBuilder();

    while (value >= magnitude)
    {
        value -= magnitude;
        numberString.Append(letter);
    }
    return(numberString.ToString());
}

public static implicit operator string(
RomanNumeral roman)
{
    int        temp = roman.value;

    StringBuilder retval = new StringBuilder();

    retval.Append(RomanNumeral.NumberString(ref temp, 1000, 'M'));
    retval.Append(RomanNumeral.NumberString(ref temp, 500, 'D'));
    retval.Append(RomanNumeral.NumberString(ref temp, 100, 'C'));
    retval.Append(RomanNumeral.NumberString(ref temp, 50, 'L'));
    retval.Append(RomanNumeral.NumberString(ref temp, 10, 'X'));
    retval.Append(RomanNumeral.NumberString(ref temp, 5, 'V'));
    retval.Append(RomanNumeral.NumberString(ref temp, 1, 'I'));

    return(retval.ToString());
}
```

```
    public short ToShort()
    {
        return((short) this);
    }
    public override string ToString()
    {
        return((string) this);
    }

    private short value;
}
```

The `ToString()` function is an override because it overrides the `ToString()` version in `object`.

How It Works

To finish the section on user-defined conversions, a few details on how the compiler views conversions warrant a bit of explanation. Those who are really interested in the gory details can find them in the C# Language Reference.[2]

You can safely skip the following sections if you'd like.

Conversion Lookup

When looking for candidate user-defined conversions, the compiler will search the source class and all of its base classes and the destination class and all of its base classes.

This leads to an interesting case:

```
public class S
{
    public static implicit operator T(S s)
{
// conversion here
return(new T());
}
}

public class TBase
{
}

public class T: TBase
{

}
```

2. You can find the C# Language Reference at `http://msdn.microsoft.com/net/ecma`.

```
public class Test
{
    public static void Main()
    {
        S myS = new S();
        TBase tb = (TBase) myS;
    }
}
```

In this example, the compiler will find the conversion from `S` to `T` and, since the use is explicit, match it for the conversion to `TBase`, which will succeed only if the `T` returned by the conversion is really only a `TBase`.

Revising things a bit by removing the conversion from `S` and adding it to `T`, you get this:

```
// error
class S
{
}
class TBase
{
}
class T: TBase
{
    public static implicit operator T(S s)
    {
        return(new T());
    }
}
class Test
{
    public static void Main()
    {
        S myS = new S();
        TBase tb = (TBase) myS;
    }
}
```

This code doesn't compile. The conversion is from `S` to `TBase`, and the compiler can't find the definition of the conversion, because it doesn't search class `T`.

CHAPTER 26

Operator Overloading

Operator overloading allows operators to be defined on a class or struct so it can be used with operator syntax. This is most useful on data types where there's a good definition for what a specific operator means, thereby allowing an economy of expression for the user.

Chapter 29 covers overloading the relational operators (==, !=, >, <, >=, and <=).

Chapter 25 covers overloading conversion operators.

Unary Operators

All unary operators are defined as static functions that take a single operator of the class or struct type and return an operator of that type. You can overload the following operators:

```
+-!~++--true false
```

The first six unary overloaded operators are called when the corresponding operation is invoked on a type. The `true` and `false` operators are available for Boolean types where `if (a == true)` isn't equivalent to the following:

```
if (! (a == false))
```

This happens in the SQL types in the `System.Data.SQL` namespace, which have a null state that's neither true nor false. In this case, the compiler will use the overloaded `true` and `false` operators to correctly evaluate such statements. These operators must return type `bool`.

There's no way to discriminate between the before and after increment or decrement operations. Because the operators are static (and therefore have no state), this distinction isn't important.

Binary Operators

All binary operators take two parameters, at least one of which must be the class or struct type in which the operator is declared. A binary operator can return any type but will typically return the type of the class or struct in which it's defined.

You can overload the following binary operators:

```
+-*/%&|^<<>>==   !=   >=   <=   >   <
```

An Example

The following class implements some of the overloadable operators:

```
using System;
struct RomanNumeral
{
    public RomanNumeral(int value)
    {
        this.value = value;
    }
    public override string ToString()
    {
        return(value.ToString());
    }
    public static RomanNumeral operator -(RomanNumeral roman)
    {
        return(new RomanNumeral(-roman.value));
    }
    public static RomanNumeral operator +(
    RomanNumeral    roman1,
    RomanNumeral    roman2)
    {
        return(new RomanNumeral(
                    roman1.value + roman2.value));
    }

    public static RomanNumeral operator ++(
    RomanNumeral    roman)
    {
        return(new RomanNumeral(roman.value + 1));
    }
    int value;
}
class Test
{
    public static void Main()
    {
        RomanNumeral    roman1 = new RomanNumeral(12);
        RomanNumeral    roman2 = new RomanNumeral(125);

        Console.WriteLine("Increment: {0}", roman1++);
        Console.WriteLine("Addition: {0}", roman1 + roman2);
    }
}
```

This example generates the following output:

```
Increment: 12
Addition: 138
```

Restrictions

It isn't possible to overload member access, member invocation (function calling), or the =, &&, ||, ?:, or new operators. This is for the sake of simplicity; although you can do interesting things with such overloadings, it greatly increases the difficulty of understanding the code, since you have to always remember that member invocation (for example) could be doing something special.[1] The new operator can't be overloaded because the .NET runtime is responsible for managing memory, and in the C# idiom, new just means "give me a new instance of."

It's also not possible to overload the compound assignment operators (+=, *=, and so on), since they're always expanded to the simple operation and an assignment. This avoids cases where one is defined and the other isn't, or cases where (shudder) they're defined with different meanings.

Guidelines

Operator overloading is a feature you should use only when necessary. In other words, use it only when it makes things easier for the user.

A good example of operator overloading is defining arithmetic operations on a complex number or matrix class.

A bad example is defining the increment (++) operator on a string class to mean "increment each character in the string." A good guideline is that unless a typical user would understand what the operator does without any documentation, you shouldn't define it as an operator. Don't make up new meanings for operators.

In practice, the equality (==) and inequality (!=) operators are the ones you'll define most often, since if you don't do this, you may get unexpected results.[2]

If the type behaves like a built-in data type, such as the BinaryNumeral class, it may make sense to overload more operators. At first look, it might seem that since the BinaryNumeral class is really just a fancy integer, it could just derive from the System.Int32 class and get the operators for free.

This won't work for a couple of reasons. First, you can't use value types as base classes, and Int32 is a value type. Second, even if it's possible, it won't really work for BinaryNumeral, because a BinaryNumeral isn't an integer; it supports only a small part of the possible integer range. Because of this, derivation isn't a good design choice. The smaller range means that even if BinaryNumeral is derived from int, there isn't an implicit conversion from int to BinaryNumeral, and any expressions therefore require casts.

---

1. You could, however, argue that member access can be overloaded through properties.
2. As you saw earlier, if your type is a reference type (class), using == will check if the two things you're comparing reference the same object, rather than seeing if they have the same contents. If your type is a value type, == will compare the contents of the value type, if operator == is defined.

Even if these facts weren't true, however, it still wouldn't make sense, since the whole point of having a data type is to have something that's lightweight, and a struct would be a better choice than a class. Structs, of course, can't derive from other objects.

A Complex Number Class

The following struct implements a complex number, with a few overloaded operators. Note that there are nonoverloaded versions for languages that don't support overloaded operators.

```
using System;

struct Complex
{
    float real;
    float imaginary;

    public Complex(float real, float imaginary)
    {
        this.real = real;
        this.imaginary = imaginary;
    }

    public float Real
    {
        get
        {
            return(real);
        }
        set
        {
            real = value;
        }
    }

    public float Imaginary
    {
        get
        {
            return(imaginary);
        }
        set
        {
            imaginary = value;
        }
    }
```

```
public override string ToString()
{
    return(String.Format("({0}, {1}i)", real, imaginary));
}

public static bool operator==(Complex c1, Complex c2)
{
    if ((c1.real == c2.real) &&
        (c1.imaginary == c2.imaginary))
        return(true);
    else
        return(false);
}

public static bool operator!=(Complex c1, Complex c2)
{
    return(!(c1 == c2));
}

public override bool Equals(object o2)
{
    Complex c2 = (Complex) o2;

    return(this == c2);
}

public override int GetHashCode()
{
    return(real.GetHashCode() ^ imaginary.GetHashCode());
}

public static Complex operator+(Complex c1, Complex c2)
{
    return(new Complex(c1.real + c2.real, c1.imaginary + c2.imaginary));
}

public static Complex operator-(Complex c1, Complex c2)
{
    return(new Complex(c1.real - c2.real, c1.imaginary - c2.imaginary));
}

    // product of two complex numbers
public static Complex operator*(Complex c1, Complex c2)
{
    return(new Complex(c1.real * c2.real - c1.imaginary * c2.imaginary,
                c1.real * c2.imaginary + c2.real * c1.imaginary));
}
```

```
        // quotient of two complex numbers
    public static Complex operator/(Complex c1, Complex c2)
    {
        if ((c2.real == 0.0f) &&
            (c2.imaginary == 0.0f))
            throw new DivideByZeroException("Can't divide by zero Complex number");

        float newReal =
            (c1.real * c2.real + c1.imaginary * c2.imaginary) /
            (c2.real * c2.real + c2.imaginary * c2.imaginary);
        float newImaginary =
            (c2.real * c1.imaginary - c1.real * c2.imaginary) /
            (c2.real * c2.real + c2.imaginary * c2.imaginary);

        return(new Complex(newReal, newImaginary));
    }

        // non-operator versions for other languages...
    public static Complex Add(Complex c1, Complex c2)
    {
        return(c1 + c2);
    }

    public static Complex Subtract(Complex c1, Complex c2)
    {
        return(c1 - c2);
    }

    public static Complex Multiply(Complex c1, Complex c2)
    {
        return(c1 * c2);
    }

    public static Complex Divide(Complex c1, Complex c2)
    {
        return(c1 / c2);
    }
}

class Test
{
    public static void Main()
    {
        Complex c1 = new Complex(3, 1);
        Complex c2 = new Complex(1, 2);

        Console.WriteLine("c1 == c2: {0}", c1 == c2);
```

```
        Console.WriteLine("c1 != c2: {0}", c1 != c2);
        Console.WriteLine("c1 + c2 = {0}", c1 + c2);
        Console.WriteLine("c1 - c2 = {0}", c1 - c2);
        Console.WriteLine("c1 * c2 = {0}", c1 * c2);
        Console.WriteLine("c1 / c2 = {0}", c1 / c2);
    }
}
```

CHAPTER 27

Nullable Types

Null values are a useful programming construct beyond simply recording the initial state of a class variable. In the database world, null conveys a notion of "no value recorded." In a survey, the answer to a question may be *yes* or *no*, but if a response isn't provided, isn't available, or isn't relevant to a particular respondent, you can record their response as *null* rather than a definite *yes* or *no*. Because many applications deal with data that's stored in databases, where null values can be quite common, the ability to express null values becomes quite important.

The inability to store nulls in value types can be a significant problem in C#. Value types, by definition, consist of bits that either are allocated on the stack when they're declared as a local variable or are allocated inline when they form part of heap-allocated types. Because a value type is always physically allocated, it has no ability to express an unallocated or null value type. Further, since value types don't have the luxury of an object header (like reference types do), they don't have a spare header bit that can indicate an instance is null.

Dealing with the inability to express nullability becomes particularly troublesome when dealing with data that originates from databases. Databases typically don't have separate concepts for reference and value types, and all data types can be null if the schema allows. Before nullable types, recording null values in value types was a difficult task. The following options existed:

- You could choose a special value to indicate logical null. For signed integers, `-1` was often a reasonable choice; for floating points, NaN was popular. Finally, for dates, `DateTime.MinValue` was often used.
- You could use a reference type that stored the value type internally and leave it as null when appropriate. The types in `System.Data.SqlTypes` and various third-party libraries were used in this role.
- You could maintain a Boolean variable for every value type variable that could potentially be null. The Boolean would keep track of whether the accompanying value type was logically null.

All of these alternatives had problems. Special values were often hard to define and had the unfortunate tendency to end up back in the database if a developer wasn't careful with the update logic. This in turn created the painful problem of catering to both physical and logical nulls. Using reference types was effective, but this data-tier local implementation decision had a tendency of spreading through all tiers of an application, which meant that the performance benefits offered by value types weren't available. The third alternative was generally the cleanest

and simplest option, but it was cumbersome to implement, particularly with method return values, and it also wasted space.

Because developers needed a general-purpose solution to value type nullability, and because the designers of C# 2.0 were empowered by the addition of generics, they extended the framework library to include the `Nullable<T>` value type. `Nullable<T>` can take any value type as a type parameter, and it internally stores a Boolean to track logical nullness. This approach is essentially the same as the third option from the previous list but is simpler from a logistical point of view, as only a single variable needs to be tracked.

`Nullable<T>` has various retrieval and informational methods:

```
Nullable<int> i = null;
bool b = i.HasValue; //will be false
int j = Nullable.GetValueOrDefault<int>(i); //returns 0
//j = i.Value; //will throw an exception
//j = (int)i; //will throw an exception
i = 123;
j = Nullable.GetValueOrDefault<int>(i); //returns 123
j = (int)123; //works fine
```

As you can see in the example, `Nullable<T>` allows both `null` and values that are legal for the type parameter to be assigned to a `Nullable<T>` instance. You can use the `GetValueOrDefault` method to retrieve the value, with the default value for the value type returned if the instance is null. You can also use a cast to convert to a non-nullable instance, but an exception will occur if the cast instance is logically null.

C# Language Nullable Types

The designers of the C# language figured the ability to convey nullability in value types was important enough to elevate nullable types from a generic type in the framework library into a fully fledged language feature. C# language nullable types map directly to the `Nullable<T>` framework library type, so any other language that supports generics can use C# nullable types. You express nullable types by adding a question mark after the name of a value type, which indicates that the instance declared by the declaration can be nullable:

```
int? i = null; //same as Nullable<int> i = null;
int? j = 0; // same as Nullable<int> j = 0;
```

To be a genuinely useful feature, nullable types need to act and feel like the value type they represent. Thankfully, operations and conversion that work on the value type will also work on the reference type. This means the following code will work with no compile-time or runtime errors:

```
int? i = 1;
int? j = 2;
int? k = i + j;  //addition operator works fine
double? d = k; //no problem with implicit conversion to double
short? s = (short?)d; //explicit cast required to short?
                      //because of possible data loss
```

For the case where none of the values in the equation is null, life is simple: the operations and conversions behave in the same way as they would have if the values hadn't been nullable types. When null values are present, the situation becomes a little more interesting. The basic rule is that the presence of a null value anywhere in the chain results in a null result. Take the following example:

```
int? i = 1;
int? h = 2;
int? j = null;
int? k = i + j + h + 3;
double? d = k;
short? s = (short?)d;
```

In this case, the fact that `j` is null and is used in the addition results in `k`, `d`, and `s` ending up null.

User-defined operators work in much the same way as the built-in operators. If no null values are present, the user-defined operations work in the same way as they would with the non-nullable form of the types. If one or more null values are present anywhere in the chain of operations, the result of the user-defined operation will be null.

SQL Language Differences and Similarities

In the C language, two null pointers are equal, as pointer comparison is simply a comparison of two integral values. C# follows this pattern, and for equality operations, two null values *are* defined as being equal, a behavior that's in contrast to SQL logic where null isn't equal to null but instead *is* null.

During the C# 2.0 design process, the designers discussed this decision at length and considered many options. One option was to support both the programming and database versions of equality (`null == null` and `null != null`), but this would have required an additional set of operators. The C# design team ultimately decided that approach would add too much complexity. They decided using the database version of equality in code would be surprising to the majority of C# programmers.

The following code highlights the different behavior between the two languages:

```
//C#
int? i = null;
int? j = null;
bool b = i == j;
//b with be true


--T_SQL
declare @i int
declare @j int
select @i = null
select @j = null
```

```
declare @b bit
if @i = @j
 select @b = 1
else
 select @b = 0
--@b will be zero.
```

In contrast to the difference in behavior between C# and SQL nullable types, significant efforts have been made to ensure compatibility in several areas. The nullable `bool` type has predefined operators for bitwise AND (`&`) and OR (`|`) operations that ensure it has the same behavior as the equivalent SQL operations. This means a bitwise operation on a `bool?` instance doesn't necessary result in a null result, even if one of the inputs into the operation is null. Clearly, a major exception to the main rule exists—a null anywhere in a calculation chain results in the outcome of the calculation being null. The following are a few examples:

```
bool? i = true;
bool? j = false;
bool? k = null;

bool? res1 = i | k; //returns true, NOT null
bool? res2 = j & k; //returns false, NOT null
```

Null Coalescing Operator

Just as the C# language provides a more streamlined syntax for expressing nullable types, a more streamlined syntax exists for providing a default alternative if a nullable type doesn't hold a value. Rather than using the `HasValue` method and the `?:` ternary conditional operator to provide a default alternative, you can use the null coalescing operator (`??`)—simply place the default value after the `??` operator:

```
int? x = null;
int y = x ?? 2; //x is null, y will be 2
//same as:
y = x.HasValue ? x. GetValueOrDefault() : 2;

x = 3;
y = x ?? 2; //x has a value and it will be returned, y will be 3
```

You have no need to cast the nullable type instance when using the coalescing operator; the compiler will generate a call to `GetValueOrDefault`, which returns an instance of the non-nullable value type.

Design Guidelines

When user interfaces and data tiers are written to present and manage data that's persisted to a database, nullable types make the exercise of dealing with value types that can be null much easier. To achieve the smoothest development process with database code, one of the essential elements is a consistent and reliable approach for mapping database types to C# types and

dealing with the changes, additions, and deletions in these types that inevitably occur during the development cycle.

Because nullability is one of a number of attributes of any particular database field, and because dealing with changes in nullability is no different from dealing with name and data type changes, the idea of making a value type nullable in C# when it isn't nullable in the database just in case the underlying database schema changes makes little sense. If the nullability of a field changes, simply use the same established change process you'd use if an `int` field changed to a `long` field. Using nullable types unnecessarily is detrimental to performance—memory is wasted storing a Boolean value for each nullable type and processor cycles are wasted checking this Boolean when calculations involving the nullable type are performed.

Don't mix nullability concepts within an assembly. If you're currently using special values, `System.Data.SqlTypes` types, or third-party nullability libraries to deal with nullability, don't introduce `Nullable<T>` or C# nullable types without migrating existing code to use these new features.

CHAPTER 28

Other Language Details

This chapter covers some miscellaneous details about the language, including how to use the `Main()` function, how the preprocessor works, and how to write literal values.

The Main Function

The simplest version of the `Main()` function will already be familiar to you from other examples:

```
using System;
class Test
{
    public static void Main()
    {
        Console.WriteLine("Hello, Universe!");
    }
}
```

Returning an int Status

It's often useful to return a status from the `Main()` function, particularly if the program is called programmatically; this is because you can use the return status to determine if the application executed. You do this by declaring the return type of `Main()` as an integer:

```
using System;
class Test
{
    public static int Main()
    {
        Console.WriteLine("Hello, Universe!");
        return(0);
    }
}
```

Command-Line Parameters

You can access the command-line parameters to an application by declaring the `Main()` function with a `string` array as a parameter, as in the following code. You can then process the parameters by indexing the array.

```
using System;
class Test
{
    public static void Main(string[] args)
    {
        foreach (string arg in args)
            Console.WriteLine("Arg: {0}", arg);
    }
}
```

Multiple Main() Functions

It's often useful for testing purposes to include a `static` function in a class that tests the class to make sure it does the right thing. In C#, you can write this `static` test function as a `Main()` function, which makes automating such tests easy.

If the compiler encounters a single `Main()` function during a compilation, the C# compiler will use it. If more than one `Main()` function exists, you should specify the class that contains the desired `Main()` on the command line with the `/main:<classname>` option.

```
// error
using System;
class Complex
{
    static int Main()
    {
        // test code here
        Console.WriteLine("Console: Passed");
        return(0);
    }
}
class Test
{
    public static void Main(string[] args)
    {
        foreach (string arg in args)
            Console.WriteLine(arg);
    }
}
```

Compiling this file with `/main:Complex` will use the test version of `Main()`, whereas compiling with `/main:Test` will use the real version of `Main()`. Compiling it without either will result in an error.

The `Main()` declared in the `Complex` type isn't declared `public`. In fact, it has no requirement that `Main()` should be public; keeping it private is useful in cases such as these where the test function shouldn't be visible to users of the class.

Preprocessing

The most important thing to remember about the C# preprocessor is that it doesn't exist. The features from the C/C++ processor are either totally absent or present only in a limited form. In the absent category are `include` files and the ability to do text replacement with `#define`. The `#ifdef` and associated directives are present and can control the compilation of code.

Getting rid of the macro version of `#define` allows you to understand more clearly what the program is saying. A name that isn't familiar must come from one of the namespaces, and you don't have to hunt through `include` files to find it.

One reason for this change is that getting rid of preprocessing and `#include` enables a simplified compilation structure, and therefore you get some impressive improvements in compilation speed.[1] Additionally, you don't need to write a separate header file and keep it in sync with the implementation file.

Getting rid of `#define` also ensures that C# code always means what it says and that no time is wasted hunting in `include` files for macro definitions.

When C# source files are compiled, the order of the compilation of the individual files is unimportant, and it's equivalent to them all being in one big file. You don't need forward declarations or have to worry about the order of `#includes`.

Preprocessing Directives

Table 28-1 lists the preprocessing directives that are supported.

Table 28-1. *C# Preprocessing Directives*

Directive	Description
`#define` identifier	Defines an identifier. Note that a value can't be set for it; it can merely be defined. Identifiers can also be defined via the command line.
`#undef` identifier	Undefines an identifier.
`#if` expression	Code in this section is compiled if the expression is true.
`#elif` expression	This is an `else-if` construct. If the previous directive wasn't taken and the expression is true, code in this section is compiled.
`#else`	If the previous directive wasn't taken, code in this section is compiled.
`#endif`	Marks the end of a section.

1. When we first installed a copy of the compiler on our system, we typed in a simple example and compiled it, and it came back fast—so fast that we were convinced something was wrong and hunted down a developer for assistance. It's *so* much faster than C++ compilers are (or can be).

Here's an example of how you can use these directives:

```
#define DEBUGLOG
using System;
class Test
{
    public static void Main()
    {
        #if DEBUGLOG
        Console.WriteLine("In Main - Debug Enabled");
        #else
        Console.WriteLine("In Main - No Debug");
        #endif
    }
}
```

`#define` and `#undef` must precede any "real code" in a file; otherwise, an error occurs. For example, you can't write the previous example as follows:

```
// error
using System;
class Test
{
    #define DEBUGLOG
    public static void Main()
    {
        #if DEBUGLOG
        Console.WriteLine("In Main - Debug Enabled");
        #else
        Console.WriteLine("In Main - No Debug");
        #endif
    }
}
```

C# also supports the `Conditional` attribute for controlling function calls based upon preprocessor identifiers; see Chapter 39 for more information.

Preprocessor Expressions

You can use the operators described in Table 28-2 in preprocessor expressions.

Table 28-2. *C# Preprocessor Expressions*

Operator	Description
`! ex`	True if `ex` is false
`ex == value`	True if `ex` is equal to `value`
`ex != value`	True if `ex` isn't equal to `value`
`ex1 && ex2`	True if both `ex1` and `ex2` are true
`ex1 || ex2`	True if either `ex1` or `ex2` is true

You can use parentheses to group expressions:

```
#if !(DEBUGLOG && (TESTLOG || USERLOG))
```

If TESTLOG or USERLOG is defined and DEBUGLOG is defined, then the expression within the parentheses is true, which is then negated by the !.

Other Preprocessor Functions

In addition to the #if and #define functions, you can use a few other preprocessor functions.

#warning and #error

#warning and #error allow warnings or errors to be reported during the compilation process. All text following the #warning or #error will be output when the compiler reaches that line.

For a section of code, you could do the following:

```
#warning Check algorithm with John
```

This results in the string "Check algorithm with John" being output when the line is compiled.

#line

With #line, you can specify the name of the source file and the line number that are reported when the compiler encounters errors. You'd typically use this with machine-generated source code so you could sync the reported lines with a different naming or numbering system. You can also use this directive to hide lines from the debugger, as discussed in the next section.

#region

You can use the #region and corresponding #endregion directives to define a block of code that can be collapsed and expanded inside the code editor. In .NET 1.0 and 1.1, one of the main uses of the #region directive was to separate machine-generated code from human-generated code, but partial classes (see Chapter 8) have largely solved this problem.

Using the #region directive doesn't prevent the debugger from stepping into a code block. To accomplish this, you also need the #line directive:[2]

```
class Program
{
 static void Main(string[] args)
 {

  #region Logging Code
#line hidden
//debugger will not hit this line
  Console.WriteLine("In Program.Main");
  #line default
  #endregion

  Console.WriteLine("Main app stuff here");
 }
}
```

2. Thanks to the great folks at Readify.net for this tip.

Inline Warning Control

The C# compiler provides the ability to suppress warnings using the /nowarn switch. This switch, which is covered in more detail in Chapter 41, is a setting that affects all the source code files processed as part of a compile, and a per-source code file setting is often more desirable. For various reasons, living with compiler warnings is often a fact of life in real-world applications, but using the compiler switch means new warnings of the same warning number aren't reported to the developer. Deciding to tolerate the warning output is often no better, as an increase in the number of warnings from, say, 116 to 117 is much less noticeable than the warning count jumping from 0 to 1.

C# 2.0 augments the global compiler switch with a new #pragma warning directive that allows you to control warning production at a granular level. At a basic level, you can use a #pragma warning disable XXX statement (where *XXX* is the number of the error being suppressed) to suppress all warnings of that number in the source file. Accompanying the warning suppression with a comment indicating why the error is unavoidable or irrelevant in this instance will help the maintenance programmer significantly and also jog your memory. For example:

```
class Program
{
 static void Main(string[] args)
 {
  //Use the original DoStuff method (now marked Obsolete) until the bug
  //in new version that is documented in vendor KB article 123 is fixed.
#pragma warning disable 612
  DoStuff();
#pragma warning restore 612
 }

 [Obsolete]
 static void DoStuff() {}
}
```

In this case, the #pragma warning restore statement has restored the warning, which means if it occurs again in the same source file and outside the scope of the #pragma region, it will generate a warning.

You can disable multiple warnings in the same line by placing commas between the warning numbers:

```
//disable Obsolete and GetHashCode warnings
#pragma warning disable 612, 661
```

The ability to control warnings at such a granular level, combined with the existing ability to globally disable particular warnings (missing XML comments in third-party libraries often leads to a global /nowarn on 1591), means that for production-quality code, having zero warnings is an attainable and desirable goal. Additionally, depending on the development methodology and the formality of the development process, beginning with or progressing to the practice of treating warnings as errors (using the /warnaserror compiler switch) is recommended.

Lexical Details

The lexical details of the language specify features that are important at the single-character level: how to write numerical constants, identifiers, and other low-level entities of the language.

Identifiers

An *identifier* is a name that's used for some program element, such as a variable or a function.

Identifiers must have a letter or an underscore as the first character, and the remainder of the identifier can also include numeric characters.[3] You can specify Unicode characters using `\udddd`, where `dddd` specifies the hex value of the Unicode character.

When using code that has been written in other languages, some names might be C# keywords. To write such a name, place an at (`@`) sign before the name, which merely indicates to C# that the name isn't a keyword but an identifier.

Similarly, use `@` to implement keywords as identifiers:

```
class Test
{
    public void @checked()
    {
    }
}
```

This class declares a member function named `checked`.

Using this feature so that identifiers can be the same as built-in identifiers isn't recommended because of the confusion it can create.

Keywords

Keywords are reserved words that can't be used as identifiers. The following are the keywords in C#:

abstract	as	base	bool	break
byte	case	catch	char	checked
class	const	continue	decimal	default
delegate	do	double	else	enum
event	explicit	extern	false	finally
fixed	float	for	foreach	goto
if	implicit	in	int	interface
internal	is	lock	long	namespace

3. It's actually a fair bit more complicated than this, since C# has Unicode support. Briefly, letters can be any Unicode letter character, and you can use characters other than the underscore (_) for combinations. See the C# Language Reference (`http://msdn.microsoft.com/net/ecma`) for a full description.

new	null	object	operator	out
override	params	private	protected	public
readonly	ref	return	sbyte	sealed
short	sizeof	stackalloc	static	string
struct	switch	this	throw	true
try	typeof	uint	ulong	unchecked
unsafe	ushort	using	virtual	volatile
void	while			

Also, four keywords (`partial`, `where`, `yield`, and `value`) are contextual keywords. You can use a contextual keyword as a variable, class, or property name, but when you use it in a specific context, it becomes a keyword. You use `partial` in the specification of partial class, `where` to define generic constraints, `yield` to implement an iterator, and `value` in a `set` property block to access the incoming value.

Literals

Literals are the way in which values are written for variables.

Boolean

Two boolean literals exist: `true` and `false`.

Integer

You can write integer literals simply by writing the numeric value. Integer literals small enough to fit into the `int` data type[4] are treated as `int`s; if they're too big to fit into an `int`, they will be created as the smallest type of `uint`, `long`, or `ulong` in which the literal will fit.

Here are some integer literal examples:

```
123
-15
```

You can also write integer literals in hexadecimal format by placing `0x` in front of the constant:

```
0xFFFF
0x12AB
```

Real

Real literals are for the types `float`, `double`, and `decimal`. Float literals have `f` or `F` after them; double literals have `d` or `D` after them and are the default when nothing is specified, and `decimal` literals have `m` or `M` after them.

You can use exponential notation by appending `e`, followed by the exponent to the real literal.

4. See Chapter 3 for more information.

Here are some examples:

```
1.345                   // double constant
-8.99e12F               // float constant
15.66m                  // decimal constant
```

Character

A character literal is a single character enclosed in single quotes, such as `'x'`. The escape sequences shown in Table 28-3 are supported.

Table 28-3. *Character Escape Sequences*

Escape Sequence	Description
`\'`	Single quote. Single quotes don't need to be escaped if they're part of a character literal.
`\"`	Double quote.
`\\`	Backslash.
`\0`	Null.
`\a`	Alert.
`\b`	Backspace.
`\f`	Form feed.
`\n`	Newline.
`\r`	Carriage return.
`\t`	Tab.
`\v`	Vertical tab.
`\xdddd`	Character `dddd`, where `d` is a hexadecimal digit.
`\udddd`	Unicode character `dddd`, where `d` is a hexadecimal digit.
`\Uddddddddd`	Unicode character `dddddddd`, where `d` is a hexadecimal digit.

String

You write string literals as a sequence of characters enclosed in double quotes, such as `"Hello"`. All the character escape sequences are supported within strings.

Strings can't span multiple lines, but you can achieve the same effect by concatenating them:

```
string s = "What is your favorite color?" +
             "Blue. No, Red. ";
```

When this code is compiled, a single string constant will be created that consists of the two strings concatenated.

Verbatim Strings

Verbatim strings allow you to specify some strings more simply.

If a string contains the backslash character, such as a filename, you can use a verbatim string to turn off the support for escape sequences. Instead of writing something like this:

```
string s = "c:\\Program Files\\Microsoft Office\\Office";
```

you can write the following:

```
string s = @"c:\Program Files\Microsoft Office\Office";
```

The verbatim string syntax is also useful if the code is generated by a program and you have no way to constrain the contents of the string. You can represent all characters within such a string, but you must double any occurrence of a double quote:

```
string s = @"She said, ""Hello""";
```

In addition, strings that are written with the verbatim string syntax can span multiple lines, and you can preserve whitespace (spaces, tabs, and newlines):

```
using System;
class Test
{
    public static void Main()
    {
        string s = @"
        C: Hello, Miss?
        O: What do you mean, 'Miss'?
        C: I'm Sorry, I have a cold. I wish to make a complaint.";
        Console.WriteLine(s);
    }
}
```

Comments

Comments in C# are denoted by a double slash for a single-line comment and by /* and */ for the beginning and ending of a multiline comment:

```
// This is a single-line comment
/*
 * Multiline comment here
 */
```

C# also supports a special type of comment that associates documentation with code; Chapter 38 describes those comments.

CHAPTER 29

Making Friends with the .NET Framework

The information in the preceding chapters is sufficient for writing objects that will function in the .NET runtime, but those objects won't feel like they were written to operate well in the .NET Framework. This chapter details how to make user-defined objects operate more like the objects in the .NET runtime and the .NET Framework.

Things All Objects Will Do

Overriding the `ToString()` function from the `object` class gives a nice representation of the values in an object. If this isn't done, `object.ToString()` will merely return the name of the class.

The `Equals()` function on `object` is called by the .NET Framework classes to determine whether two objects are equal.

A class may also override `operator==()` and `operator!=()`, which allows the user to use the built-in operators with instances of the object, rather than calling `Equals()`.

`ToString()`

Here's an example of what happens by default:

```
using System;
public class Employee
{
    public Employee(int id, string name)
    {
        this.id = id;
        this.name = name;
    }
    int id;
    string name;
}
```

```
class Test
{
    public static void Main()
    {
        Employee herb = new Employee(555, "Herb");
        Console.WriteLine("Employee: {0}", herb);
    }
}
```

The preceding code results in the following:

```
Employee: Employee
```

By overriding `ToString()`, the representation can be much more useful:

```
using System;
public class Employee
{
    public Employee(int id, string name)
    {
        this.id = id;
        this.name = name;
    }
    public override string ToString()
    {
        return(String.Format("{0}({1})", name, id));
    }
    int id;
    string name;
}
class Test
{
    public static void Main()
    {
        Employee herb = new Employee(555, "Herb");
        Console.WriteLine("Employee: {0}", herb);
    }
}
```

This gives you a far better result:

```
Employee: Herb(555)
```

When `Console.WriteLine()` needs to convert an object to a string representation, it will call the `ToString()` virtual function, which will forward to an object's specific implementation. If you desire more control over formatting, such as implementing a floating-point class with different formats, you can override the `IFormattable` interface. Chapter 34 covers `IFormattable`.

Equals()

`Equals()` determines whether two objects have the same contents. This function is called by the collection classes (such as `Array` or `Hashtable`) to determine whether two objects are equal. Extending the employee example, you can write the following:

```
using System;
public class Employee
{
    public Employee(int id, string name)
    {
        this.id = id;
        this.name = name;
    }
    public override string ToString()
    {
        return(name + "(" + id + ")");
    }
    public override bool Equals(object obj)
    {
        return(this == (Employee) obj);
    }
    public override int GetHashCode()
    {
        return(id.GetHashCode() ^ name.GetHashCode());
    }
    public static bool operator==(Employee emp1, Employee emp2)
    {
        if (emp1.id != emp2.id)
            return(false);
        if (emp1.name != emp2.name)
            return(false);
        return(true);
    }
    public static bool operator!=(Employee emp1, Employee emp2)
    {
        return(!(emp1 == emp2));
    }
    int id;
    string name;
}
```

```
class Test
{
    public static void Main()
    {
        Employee herb = new Employee(555, "Herb");
        Employee herbClone = new Employee(555, "Herb");
        Console.WriteLine("Equal: {0}", herb.Equals(herbClone));
        Console.WriteLine("Equal: {0}", herb == herbClone);
    }
}
```

This produces the following output:

```
Equal: true
Equal: true
```

In this case, `operator==()` and `operator!=()` have also been overloaded, which allows the operator syntax to be used in the last line of `Main()`. These operators must be overloaded in pairs; they can't be overloaded separately.[1]

Note that in this example, the implementation of `Equals()` forwards to the operator implementation. For this example, you could do it in either way, but for structs, you'll require an extra boxing operation if you do it the other way. Because `Equals()` takes an object parameter, a value type must always be boxed to call `Equals()`, but boxing isn't required to call the strongly typed comparison operators. If the operators forwarded to `Equals()`, they'd have to box always.

Hashes and GetHashCode()

The .NET Framework includes the `Hashtable` class, which is useful for doing fast lookups of objects by a key. A hash table works by using a hash function, which produces an integer "key" for a specific instance of a class. This key is a condensed version of the contents of the instance. While instances can have the same hash code, it's fairly unlikely to happen.

A hash table uses this key as a way of drastically limiting the number of objects that must be searched to find a specific object in a collection of objects. It does this by first getting the hash value of the object, which will eliminate all objects with a different hash code, leaving only those with the same hash code to be searched. Since the number of instances with that hash code is small, searches can be much quicker.

That's the basic idea—for a more detailed explanation, please refer to a good data structures and algorithms book.[2] Hashes are a tremendously useful construct. The `Hashtable` class stores objects, so it's easy to use them to store any type.

The `GetHashCode()` function should be overridden in user-written classes because the values returned by `GetHashCode()` are required to be related to the value returned by `Equals()`. Two objects that are the same by `Equals()` must always return the same hash code.

---

1. This is required for two reasons. First, if a user uses `==`, they can expect `!=` to work as well. Second, it's required to support nullable types, for which a `== b` *doesn't* imply `!(a != b)`.
2. I've always liked *Algorithms in C, Part 1–5: Fundamentals, Data Structures, Sorting, Searching, and Graph Algorithms*, Third Edition (Addison-Wesley, 2001) by Robert Sedgewick as a good introduction.

The default implementation of `GetHashCode()` doesn't work this way, and therefore it must be overridden to work correctly. If not overridden, the hash code will be identical only for the same instance of an object, and a search for an object that's equal but not the same instance will fail.

If there's a unique field in an object, it's probably a good choice for the hash code:

```
using System;
using System.Collections;
public class Employee
{
    public Employee(int id, string name)
    {
        this.id = id;
        this.name = name;
    }
    public override string ToString()
    {
        return(String.Format("{0}({1})", name, id));
    }
    public override bool Equals(object obj)
    {
        Employee emp2 = (Employee) obj;
        if (id != emp2.id)
            return(false);
        if (name != emp2.name)
            return(false);
        return(true);
    }
    public static bool operator==(Employee emp1, Employee emp2)
    {
        return(emp1.Equals(emp2));
    }
    public static bool operator!=(Employee emp1, Employee emp2)
    {
        return(!emp1.Equals(emp2));
    }
    public override int GetHashCode()
    {
        return(id);
    }
    int id;
    string name;
}
```

```
class Test
{
    public static void Main()
    {
        Employee herb = new Employee(555, "Herb");
        Employee george = new Employee(123, "George");
        Employee frank = new Employee(111, "Frank");
        Hashtable    employees = new Hashtable();
        employees.Add(herb, "414 Evergreen Terrace");
        employees.Add(george, "2335 Elm Street");
        employees.Add(frank, "18 Pine Bluff Road");
        Employee herbClone = new Employee(555, "Herb");
        string address = (string) employees[herbClone];
        Console.WriteLine("{0} lives at {1}", herbClone, address);
    }
}
```

In the `Employee` class, the `id` member is unique, so it's used for the hash code. In the `Main()` function, several employees are created, and they're then used as the key values to store the addresses of the employees.

If there isn't a unique value, the hash code should be created out of the values contained in a function. If the `Employee` class didn't have a unique identifier but did have fields for name and address, the hash function could use those. The following shows a hash function that could be used:[3]

```
using System;
using System.Collections;
public class Employee
{
    public Employee(string name, string address)
    {
        this.name = name;
        this.address = address;
    }
    public override int GetHashCode()
    {
        return(name.GetHashCode() + address.GetHashCode());
    }
    string name;
    string address;
}
```

This implementation of `GetHashCode()` simply XORs the hash codes of the elements together and returns them.

3. This is by no means the only hash function that could be used, or even a particularly good one. See an algorithms book for information on constructing good hash functions.

Design Guidelines

Any class that overrides `Equals()` should also override `GetHashCode()`. In fact, the C# compiler will issue an error in such a case. The reason for this error is that it prevents strange and difficult-to-debug behavior when the class is used in a `Hashtable`.

The `Hashtable` class depends on the fact that all instances that are equal have the same hash value. The default implementation of `GetHashCode()`, however, returns a value that's unique on a per-instance basis. If this implementation isn't overridden, it's easy to put objects in a hash table but not be able to retrieve them.

Value Type Guidelines

The `System.ValueType` class contains a version of `Equals()` that works for all value types, but this version of `Equals()` works through reflection if bitwise quality is invalid for the type (which will be the case if the value type has references to reference types) and is therefore slow. It's therefore recommended that an implementation of `Equals()` be written for all value types.

Reference Type Guidelines

For most reference types, users will expect that == will mean reference comparison and in this case == shouldn't be overloaded, even if the object implements `Equals()`.

If the type has value semantics (something like a `String` or a `BigNum`), `operator==()` and `Equals()` should be overridden. If a class overloads + or -, that's a pretty good indication it should also override == and `Equals()`.

A subtler area of concern is how `Equals()` operates when inheritance hierarchies come into play. Consider the following example:

```
using System;
class Base
{
    int val;

    public Base(int val)
    {
        this.val = val;
    }

    public override bool Equals(object o2)
    {
        Base b2 = (Base) o2;

        return(val == b2.val);
    }

    public override int GetHashCode()
    {
        return(val.GetHashCode());
    }
}
```

```
class Derived: Base
{
    int val2;

    public Derived(int val, int val2) : base(val)
    {
        this.val2 = val2;
    }
}
class Test
{
    public static void Main()
    {
        Base b1 = new Base(12);
        Base b2 = new Base(12);
        Derived d1 = new Derived(12, 15);
        Derived d2 = new Derived(12, 25);

        Console.WriteLine("b1 equals b2: {0}", b1.Equals(b2));
        Console.WriteLine("d1 equals d2: {0}", d1.Equals(d2));
        Console.WriteLine("d1 equals b1: {0}", b1.Equals(d1));
    }
}
```

This code generates the following results:

```
b1 equals b2: True
d1 equals d2: True
b1 equals d1: True
```

The `Base` class implements `Equals()`, and it works as expected for objects of type `Base`. Classes derived directly from `object` (or from classes that don't override `Equals()`) will work fine since they will use `object.Equals()`, which compares references.

But any class derived from `Base` will inherit the implementation of `Equals()` from `Base` and will therefore generate the wrong results. Because of this, any class that derives from a class that overrides `Equals()` should also override `Equals()`.[4] You can guard against this situation by adding a check to make sure the object is the expected type:

```
public override bool Equals(object o2)
{
    if (o2.GetType() != typeof(Base) || GetType() != typeof(Base))
        return(false);
    Base b2 = (Base) o2;

    return(val == b2.val);
}
```

4. A rare case does exist where the derived class has the same concept of equality as the base class.

This gives the following output:

```
b1 equals b2: True
d1 equals d2: False
b1 equals d1: False
```

This is correct and prevents a derived class from accidentally using the base class `Equals()` accidentally. It's now obvious that `Derived` needs its own version of `Equals()`. The code for `Derived.Equals()` will use `Base.Equals()` to check whether the base objects are equal and then compare the derived fields. The code for `Derived.Equals()` looks like this:

```
public override bool Equals(object o2)
{
    if (o2.GetType() != typeof(Derived) || GetType() != typeof(Derived))
        return(false);

    Derived d2 = (Derived) o2;
    return(base.Equals(o2) && val2 == d2.val2);
}
```

Adding this code generates the following output:

```
b1 equals b2: True
d1 equals d2: False
b1 equals d1: False
```

That's clearly wrong. What's causing the problem is that the type check in the base class will always return false when called from `Derived.Equals()`.

Since it doesn't work to check for an exact type, the next best thing is to check that the types are the same. The code for `Base.Equals()` becomes the following:

```
public override bool Equals(object o2)
{
    if (o2 == null || GetType() != o2.GetType())
        return false;

    Base b2 = (Base) o2;

    return(val == b2.val);
}
```

And the code for `Derived.Equals()` uses the same check and also calls `base.Equals()`. This code also checks for `null` to prevent an exception when comparing to a null reference.

The following list summarizes this discussion:

- Reference types should make sure both types are the same in `Equals()`.
- If the type is derived from a type that overrides `Equals()`, `Base.Equals()` should be called to check whether the base portion of the type is equal.
- If the type is derived from a type that doesn't override `Equals()`, `Base.Equals()` shouldn't be called since it'd be `object.Equals()`, which implements reference comparison.

CHAPTER 30

System.Array and the Collection Classes

Conceptually, this chapter gives an overview of what classes are available. It then covers them by class and gives examples of what interfaces and functions are required to enable specific functionality. Finally, the chapter covers the new generic collections.

Sorting and Searching

The .NET Framework collection classes provide some useful support for sorting and searching, with built-in functions that do sorting and binary searching. The `Array` class provides the same functionality, but as static functions rather than as member functions.

Sorting an array of integers is as easy as this:

```
using System;
class Test
{
    public static void Main()
    {
        int[]    arr = {5, 1, 10, 33, 100, 4};
        Array.Sort(arr);
        foreach (int v in arr)
            Console.WriteLine("Element: {0}", v);
    }
}
```

The preceding code gives the following output:

```
4
5
10
33
100
```

This is convenient for the built-in types, but it doesn't work for classes or structs because the sort routine doesn't know how to order them.

Implementing IComparable

The .NET Framework has some nice ways for a class or struct to specify how to order instances of the class or struct. In the simplest one, the object implements the `IComparable` interface:

```
using System;
public class Employee: IComparable
{
    public Employee(string name, int id)
    {
        this.name = name;
        this.id = id;
    }

    int IComparable.CompareTo(object obj)
    {
        Employee emp2 = (Employee) obj;
        if (this.id > emp2.id)
            return(1);
        if (this.id < emp2.id)
            return(-1);
        else
            return(0);
    }

    public override string ToString()
    {
        return(String.Format("{0}:{1}", name, id));
    }

    string   name;
    int    id;
}
class Test
{
    public static void Main()
    {
        Employee[] arr = new Employee[4];
        arr[0] = new Employee("George", 1);
        arr[1] = new Employee("Fred", 2);
        arr[2] = new Employee("Tom", 4);
        arr[3] = new Employee("Bob", 3);
```

```
        Array.Sort(arr);
        foreach (Employee emp in arr)
            Console.WriteLine("Employee: {0}", emp);
            // Find employee id 2 in the list;
        Employee employeeToFind = new Employee(null, 2);
        int index = Array.BinarySearch(arr, employeeToFind);
        if (index != -1)
            Console.WriteLine("Found: {0}", arr[index]);
    }
}
```

This program gives you the following output:

```
Employee: George:1
Employee: Fred:2
Employee: Bob:3
Employee: Tom:4
Found: Fred:2
```

This sort implementation allows only one sort ordering; you could define the class to sort based on employee ID or based on name, but you have no way to allow the user to choose which sort order they prefer.

This example also uses the `BinarySearch()` method to find an employee in the list. For this to work, the array (or `ArrayList`) must be sorted, or the results won't be correct.

Using IComparer

The designers of the .NET Framework have provided the capability to define multiple sort orders. Each sort order is expressed through the `IComparer` interface, and the appropriate interface is passed to the sort or search function.

The `IComparer` interface can't be implemented on `Employee`, however, because each class can implement an interface only once, which would allow only a single sort order.[1] You need a separate class for each sort order, with the class implementing `IComparer`. The class will be simple, since all it will do is implement the `Compare()` function:

```
using System;
using System.Collections;
class Employee
{
    public string name;
}
```

1. `IComparable` *could* implement one sort order and `IComparer` another, but that would be confusing to the user.

```
class SortByNameClass: IComparer
{
    public int Compare(object obj1, object obj2)
    {
        Employee emp1 = (Employee) obj1;
        Employee emp2 = (Employee) obj2;
        return(String.Compare(emp1.name, emp2.name));
    }
}
```

The `Compare()` member takes two objects as parameters. Since the class should be used for sorting employees only, the `object` parameters are cast to `Employee`. The `Compare()` function built into `string` is then used for the comparison.

Revise the `Employee` class as follows. Place the sort-ordering classes inside the `Employee` class as nested classes.

```
using System;
using System.Collections;

public class Employee: IComparable
{
    public Employee(string name, int id)
    {
        this.name = name;
        this.id = id;
    }

    int IComparable.CompareTo(object obj)
    {
        Employee emp2 = (Employee) obj;
        if (this.id > emp2.id)
            return(1);
        if (this.id < emp2.id)
            return(-1);
        else
            return(0);
    }

    public override string ToString()
    {
        return(name + ":" + id);
    }
```

```
    public class SortByNameClass: IComparer
    {
        public int Compare(object obj1, object obj2)
        {
            Employee emp1 = (Employee) obj1;
            Employee emp2 = (Employee) obj2;

            return(String.Compare(emp1.name, emp2.name));
        }
    }

    public class SortByIdClass: IComparer
    {
        public int Compare(object obj1, object obj2)
        {
            Employee emp1 = (Employee) obj1;
            Employee emp2 = (Employee) obj2;

            return(((IComparable) emp1).CompareTo(obj2));
        }
    }

    string    name;
    int    id;
}
class Test
{
    public static void Main()
    {
        Employee[] arr = new Employee[4];
        arr[0] = new Employee("George", 1);
        arr[1] = new Employee("Fred", 2);
        arr[2] = new Employee("Tom", 4);
        arr[3] = new Employee("Bob", 3);

        Array.Sort(arr, (IComparer) new Employee.SortByNameClass());
            // employees is now sorted by name

        foreach (Employee emp in arr)
            Console.WriteLine("Employee: {0}", emp);

        Array.Sort(arr, (IComparer) new Employee.SortByIdClass());
            // employees is now sorted by id
```

```
        foreach (Employee emp in arr)
            Console.WriteLine("Employee: {0}", emp);

        ArrayList arrList = new ArrayList();
        arrList.Add(arr[0]);
        arrList.Add(arr[1]);
        arrList.Add(arr[2]);
        arrList.Add(arr[3]);
        arrList.Sort((IComparer) new Employee.SortByNameClass());

        foreach (Employee emp in arrList)
            Console.WriteLine("Employee: {0}", emp);

        arrList.Sort();    // default is by id

        foreach (Employee emp in arrList)
            Console.WriteLine("Employee: {0}", emp);
    }
}
```

The user can now specify the sort order and switch between the different sort orders as desired. This example shows how the same functions work using the `ArrayList` class, though `Sort()` is a member function rather than a static function.

IComparer As a Property

Sorting with the `Employee` class is still a bit cumbersome since the user has to create an instance of the appropriate ordering class and then cast it to `IComparer`. You can simplify this a bit further by using static properties to do this for the user:

```
using System;
using System.Collections;

public class Employee: IComparable
{
    public Employee(string name, int id)
    {
        this.name = name;
        this.id = id;
    }

    int IComparable.CompareTo(object obj)
    {
        Employee emp2 = (Employee) obj;
        if (this.id > emp2.id)
            return(1);
        if (this.id < emp2.id)
            return(-1);
```

```
    else
        return(0);
}

public static IComparer SortByName
{
    get
    {
        return((IComparer) new SortByNameClass());
    }
}

public static IComparer SortById
{
    get
    {
        return((IComparer) new SortByIdClass());
    }
}

public override string ToString()
{
    return(name + ":" + id);
}

class SortByNameClass: IComparer
{
    public int Compare(object obj1, object obj2)
    {
        Employee emp1 = (Employee) obj1;
        Employee emp2 = (Employee) obj2;

        return(String.Compare(emp1.name, emp2.name));
    }
}

class SortByIdClass: IComparer
{
    public int Compare(object obj1, object obj2)
    {
        Employee emp1 = (Employee) obj1;
        Employee emp2 = (Employee) obj2;

        return(((IComparable) emp1).CompareTo(obj2));
    }
}
```

```
    string    name;
    int    id;
}
class Test
{
    public static void Main()
    {
        Employee[] arr = new Employee[4];
        arr[0] = new Employee("George", 1);
        arr[1] = new Employee("Fred", 2);
        arr[2] = new Employee("Tom", 4);
        arr[3] = new Employee("Bob", 3);

        Array.Sort(arr, Employee.SortByName);
            // employees is now sorted by name

        foreach (Employee emp in arr)
            Console.WriteLine("Employee: {0}", emp);

        Array.Sort(arr, Employee.SortById);
            // employees is now sorted by id

        foreach (Employee emp in arr)
            Console.WriteLine("Employee: {0}", emp);

        ArrayList arrList = new ArrayList();
        arrList.Add(arr[0]);
        arrList.Add(arr[1]);
        arrList.Add(arr[2]);
        arrList.Add(arr[3]);
        arrList.Sort(Employee.SortByName);

        foreach (Employee emp in arrList)
            Console.WriteLine("Employee: {0}", emp);

        arrList.Sort();    // default is by id

        foreach (Employee emp in arrList)
            Console.WriteLine("Employee: {0}", emp);
    }
}
```

The static properties `SortByName` and `SortById` create an instance of the appropriate sorting class, cast it to `IComparer`, and return it to the user. This simplifies the user model quite a bit; the `SortByName` and `SortById` properties return an `IComparer`, so it's obvious they can be used for sorting, and all the user has to do is specify the appropriate ordering property for the `IComparer` parameter.

Overloading Relational Operators

If a class has an ordering that's expressed in `IComparable`, it may also make sense to overload the other relational operators. As with == and !=, other operators must be declared as pairs, with < and > being one pair and >= and <= being the other pair. For example:

```
using System;
public class Employee: IComparable
{
    public Employee(string name, int id)
    {
        this.name = name;
        this.id = id;
    }

    int IComparable.CompareTo(object obj)
    {
        Employee emp2 = (Employee) obj;
        if (this.id > emp2.id)
            return(1);
        if (this.id < emp2.id)
            return(-1);
        else
            return(0);
    }
    public static bool operator <(
    Employee emp1,
    Employee emp2)
    {
        IComparable    icomp = (IComparable) emp1;
        return(icomp.CompareTo (emp2) < 0);
    }
    public static bool operator >(
    Employee emp1,
    Employee emp2)
    {
        IComparable    icomp = (IComparable) emp1;
        return(icomp.CompareTo (emp2) > 0);
    }
    public static bool operator <=(
    Employee emp1,
    Employee emp2)
    {
        IComparable    icomp = (IComparable) emp1;
        return(icomp.CompareTo (emp2) <= 0);
    }
```

```
    public static bool operator >=(
    Employee emp1,
    Employee emp2)
    {
        IComparable    icomp = (IComparable) emp1;
        return(icomp.CompareTo (emp2) >= 0);
    }

    public override string ToString()
    {
        return(name + ":" + id);
    }

    string    name;
    int    id;
}
class Test
{
    public static void Main()
    {
        Employee george = new Employee("George", 1);
        Employee fred = new Employee("Fred", 2);
        Employee tom = new Employee("Tom", 4);
        Employee bob = new Employee("Bob", 3);

        Console.WriteLine("George < Fred: {0}", george < fred);
        Console.WriteLine("Tom >= Bob: {0}", tom >= bob);
    // Find employee id 2 in the list;
        Employee employeeToFind = new Employee(null, 2);
        int index = Array.BinarySearch(arr, employeeToFind);
        if (index != -1)
            Console.WriteLine("Found: {0}", arr[index]);
    }
}
```

This example produces the following output:

```
George < Fred: true
Tom >= Bob: true
```

Generic Comparison

One of the major use cases for generics is collection classes, so it comes as no surprise that the interfaces that have relevance to collections such as `IComparable` and `IComparer` have been augmented

with a generic version. The nongeneric versions described earlier in this chapter continue to exist, and the new interfaces are in a new namespace called `System.Collections.Generic`. In addition to strongly typed comparison methods, the new generic interfaces are slightly richer than their nongeneric equivalents, with `IComparable<T>` having a strongly typed `Equals` method and `IComparer<T>` having strongly typed `Equals` and `GetHashCode` methods.

Rewriting the first example in this chapter using the generic `IComparable<T>`, you get the following:

```
using System;
using System.Collections.Generic;

public class Employee : IComparable<Employee>
{
 public Employee(string name, int id)
 {
  this.name = name;
  this.id = id;
 }
 int IComparable<Employee>.CompareTo(Employee emp2)
 {
  if (this.id > emp2.id)
   return (1);
  if (this.id < emp2.id)
   return (-1);
  else
   return (0);
 }

 bool IComparable<Employee>.Equals(Employee emp2)
 {
  if (emp2 == null)
   return false;

  return id == emp2.id && name == emp2.name;
 }


 public override string ToString()
 {
  return (String.Format("{0}:{1}", name, id));
 }
 string name;
 int id;
}
```

```
class Test
{
 public static void Main()
 {
  Employee[] arr = new Employee[4];
  arr[0] = new Employee("George", 1);
  arr[1] = new Employee("Fred", 2);
  arr[2] = new Employee("Tom", 4);
  arr[3] = new Employee("Bob", 3);
  Array.Sort(arr);
  foreach (Employee emp in arr)
   Console.WriteLine("Employee: {0}", emp);
  // Find employee id 2 in the list;
  Employee employeeToFind = new Employee(null, 2);
  int index = Array.BinarySearch(arr, employeeToFind);
  if (index != -1)
   Console.WriteLine("Found: {0}", arr[index]);
 }
}
```

As expected, the output is the same as the nongeneric version, and only the type-safety and speed of the code have been improved. You can also update the second example that used `IComparer` to use generics instead:

```
using System;
using System.Collections.Generic;

public class Employee : IComparable<Employee>
{
 public Employee(string name, int id)
 {
  this.name = name;
  this.id = id;
 }

 int IComparable<Employee>.CompareTo(Employee emp2)
 {
  if (this.id > emp2.id)
   return (1);
  if (this.id < emp2.id)
   return (-1);
  else
   return (0);
 }
```

```
 bool IComparable<Employee>.Equals(Employee emp2)
 {
  if (emp2 == null)
   return false;

  return id == emp2.id && name == emp2.name;
 }

 public override string ToString()
 {
  return (name + ":" + id);
 }
 public class SortByNameClass : IComparer<Employee>
 {
  public int Compare(Employee emp1, Employee emp2)
  {
   return (String.Compare(emp1.name, emp2.name));
  }
  public bool Equals(Employee emp1, Employee emp2)
  {
   return Compare(emp1, emp2) == 0;
  }
  public int GetHashCode(Employee emp)
  {
   return emp.name.GetHashCode();
  }
 }
 public class SortByIdClass : IComparer<Employee>
 {
  public int Compare(Employee emp1, Employee emp2)
  {
   return (((IComparable<Employee>)emp1).CompareTo(emp2));
  }
  public bool Equals(Employee emp1, Employee emp2)
  {
   return Compare(emp1, emp2) == 0;
  }
  public int GetHashCode(Employee emp)
  {
   return emp.id.GetHashCode();
  }
 }
 string name;
 int id;
}
```

```
class Test2
{
 public static void Main()
 {
  Employee[] arr = new Employee[4];
  arr[0] = new Employee("George", 1);
  arr[1] = new Employee("Fred", 2);
  arr[2] = new Employee("Tom", 4);
  arr[3] = new Employee("Bob", 3);
Array.Sort<Employee>(arr, (IComparer<Employee>)
 new Employee.SortByNameClass());
  // employees is now sorted by name
  foreach (Employee emp in arr)
   Console.WriteLine("Employee: {0}", emp);
Array.Sort<Employee>(arr, (IComparer<Employee>)
 new Employee.SortByIdClass());
  // employees is now sorted by id
  foreach (Employee emp in arr)
   Console.WriteLine("Employee: {0}", emp);
  List<Employee> list = new List<Employee>();
  list.Add(arr[0]);
  list.Add(arr[1]);
  list.Add(arr[2]);
  list.Add(arr[3]);
  list.Sort((IComparer<Employee>)new Employee.SortByNameClass());
  foreach (Employee emp in list)
   Console.WriteLine("Employee: {0}", emp);
  list.Sort(); // default is by id
  foreach (Employee emp in list)
   Console.WriteLine("Employee: {0}", emp);
 }
}
```

Notice the use of the generic `List<T>` collection instead of `ArrayList`, which was used in the corresponding example earlier in this chapter. The previous listing also used the generic `Sort` method of `Array`.

As with the first generic example, the output is the same as the nongeneric equivalent. We didn't reprint the "IComparer As a Property" example here in the interest of paper conservation, but it does accompany the book's code, which is available in the Downloads section of the Apress Web site (`http://www.apress.com`).

Advanced Use of Hashes

In some situations, it may be desirable to define more than one hash code for a specific object. You could use this, for example, to allow an employee to be searched for based on the employee ID or on the employee name. You can do this by implementing the `IHashCodeProvider` interface to provide an alternate hash function, and it also requires a matching implementation of `IComparer`. These new implementations are passed to the constructor of the `Hashtable`, like so:

```
using System;
using System.Collections;

public class Employee: IComparable
{
    public Employee(string name, int id)
    {
        this.name = name;
        this.id = id;
    }

    int IComparable.CompareTo(object obj)
    {
        Employee emp2 = (Employee) obj;
        if (this.id > emp2.id)
            return(1);
        if (this.id < emp2.id)
            return(-1);
        else
            return(0);
    }
    public override int GetHashCode()
    {
        return(id);
    }
    public static IComparer SortByName
    {
        get
        {
            return((IComparer) new SortByNameClass());
        }
    }

    public static IComparer SortById
    {
        get
        {
            return((IComparer) new SortByIdClass());
        }
    }
    public static IHashCodeProvider HashByName
    {
        get
        {
            return((IHashCodeProvider) new HashByNameClass());
        }
    }
```

```
    public override string ToString()
    {
        return(name + ":" + id);
    }

    class SortByNameClass: IComparer
    {
        public int Compare(object obj1, object obj2)
        {
            Employee emp1 = (Employee) obj1;
            Employee emp2 = (Employee) obj2;

            return(String.Compare(emp1.name, emp2.name));
        }
    }

    class SortByIdClass: IComparer
    {
        public int Compare(object obj1, object obj2)
        {
            Employee emp1 = (Employee) obj1;
            Employee emp2 = (Employee) obj2;

            return(((IComparable) emp1).CompareTo(obj2));
        }
    }
    class HashByNameClass: IHashCodeProvider
    {
        public int GetHashCode(object obj)
        {
            Employee emp = (Employee) obj;
            return(emp.name.GetHashCode());
        }
    }

    string    name;
    int    id;
}
class Test
{
    public static void Main()
    {
        Employee herb = new Employee("Herb", 555);
        Employee george = new Employee("George", 123);
        Employee frank = new Employee("Frank", 111);
```

```
        Hashtable employees =
            new Hashtable(Employee.HashByName, Employee.SortByName);
        employees.Add(herb, "414 Evergreen Terrace");
        employees.Add(george, "2335 Elm Street");
        employees.Add(frank, "18 Pine Bluff Road");
        Employee herbClone = new Employee("Herb", 000);
        string address = (string) employees[herbClone];
        Console.WriteLine("{0} lives at {1}", herbClone, address);
    }
}
```

You should use this technique sparingly. It's often simpler to expose a value, such as the employee name as a property, and allow that to be used as a hash key instead.

Synchronized Collections

When a collection class—such as `ArrayList`—is created, it isn't thread-safe, because adding synchronization to such a class imposes some overhead. If you need a thread-safe version, simply call the `Synchronized()` method to get a thread-safe wrapper to the list.

In other words, you can use the following to create a thread-safe `ArrayList`:

```
ArrayList arr = ArrayList.Synchronized(new ArrayList());
```

For more information on threading and synchronization, see Chapter 31.

The generics collections don't support the static `Synchronized` method, and you should use standard synchronization techniques such as the `lock` statement; again, see Chapter 31.

Case-Insensitive Collections

To deal with strings in a case-insensitive manner, the .NET Framework provides a way to create case-insensitive versions of the `SortedList` and `Hashtable` collection classes. This support is supplied through the `CollectionsUtil` class in the `System.Collections.Specialized` namespace by calling the `CreateCaseInsensitiveSortedList()` or `CreateCaseInsensitiveHashtable()` function.

ICloneable

You can use the `object.MemberWiseClone()` function to create a clone of an object. The default implementation of this function produces a shallow copy of an object; the fields of an object are copied exactly rather than duplicated. Consider the following:

```
using System;
class ContainedValue
{
    public ContainedValue(int count)
    {
        this.count = count;
    }
    public int count;
}
```

```
class MyObject
{
    public MyObject(int count)
    {
        this.contained = new ContainedValue(count);
    }
    public MyObject Clone()
    {
        return((MyObject) MemberwiseClone());
    }
    public ContainedValue contained;
}
class Test
{
    public static void Main()
    {
        MyObject    my = new MyObject(33);
        MyObject    myClone = my.Clone();
        Console.WriteLine(    "Values: {0} {1}",
                    my.contained.count,
                    myClone.contained.count);
        myClone.contained.count = 15;
        Console.WriteLine(    "Values: {0} {1}",
                    my.contained.count,
                    myClone.contained.count);
    }
}
```

This example produces the following output:

```
Values: 33 33
Values: 15 15
```

Because the copy made by `MemberWiseClone()` is a shallow copy, the value of `contained` is the same in both objects, and changing a value inside the `ContainedValue` object affects both instances of `MyObject`.

What's needed is a deep copy, where a new instance of `ContainedValue` is created for the new instance of `MyObject`. You do this by implementing the `ICloneable` interface:

```
using System;
class ContainedValue
{
    public ContainedValue(int count)
    {
        this.count = count;
    }
    public int count;
}
```

```
class MyObject: ICloneable
{
    public MyObject(int count)
    {
        this.contained = new ContainedValue(count);
    }
    public object Clone()
    {
        Console.WriteLine("Clone");
        return(new MyObject(this.contained.count));
    }
    public ContainedValue contained;
}
class Test
{
    public static void Main()
    {
        MyObject    my = new MyObject(33);
        MyObject    myClone = (MyObject) my.Clone();
        Console.WriteLine(    "Values: {0} {1}",
                    my.contained.count,
                    myClone.contained.count);
        myClone.contained.count = 15;
        Console.WriteLine(    "Values: {0} {1}",
                    my.contained.count,
                    myClone.contained.count);
    }
}
```

This example produces the following output:

```
Values: 33 33
Values: 33 15
```

The call to `MemberWiseClone()` will now result in a new instance of `ContainedValue`, and you can modify the contents of this instance without affecting the contents of `my`.

Unlike some of the other interfaces that might be defined on an object, `ICloneable` isn't called by the runtime; it's provided merely to ensure that the `Clone()` function has the proper signature. Some objects may choose to implement a constructor that takes an instance as a parameter instead.

Other Collections

In addition to the collection classes that have already been discussed, the .NET Framework provides a number of others, as shown in Table 30-1 and Table 30-2.

Table 30-1. *Other Collections*

`BitArray`	A compact array of bit values
`Queue`	A first-in, first-out collection
`Stack`	A last-in, first-out collection
`ListDictionary`	A lightweight implementation of `IDictionary` that's faster than `Hashtable` for small lists
`StringCollection`	A collection of strings
`StringDictionary`	A hash table but with a `string` as a key instead of an `object`
`BitVector32`	A lightweight value type to allow integer or Boolean access to a 32-bit `int`

Table 30-2. *Other Generic Collections*

`List<T>`	Generic equivalent of `ArrayList`
`Queue<T>`	Generic equivalent of `Queue`
`Stack<T>`	Generic equivalent of `Stack`
`Dictionary<K, V>`	Generic equivalent of `Hashtable`
`SortedDictionary<K, V>`	Dictionary sorted by key
`LinkedList<T>`	Generic implementation of a doubly linked list

Design Guidelines

You should consider the intended use of an object when deciding which virtual functions and interfaces to implement. Table 30-3 provides guidelines for this (generic eqivalents are shown in parentheses).

Table 30-3. *Interface and Virtual Method Uses*

Object Use	Function or Interface
General	`ToString()`
Arrays or collections	`Equals()`, `operator==()`, `operator!=()`, `GetHashCode()`
Sorting or binary search	`IComparable (IComparable<T>)`
Multiple sort orders	`IComparer (IComparer<T>)`
Multiple hash lookups	`IHashCodeProvider`

Functions and Interfaces by Framework Class

Tables 30-4, 30-5, 30-6, and 30-7 summarize which functions or interfaces on an object are used by each collection class. The generic eqivalents are shown in parentheses.

Table 30-4. *Array Interface and Virtual Method Uses*

Function	Uses
IndexOf()	Equals()
LastIndexOf()	Equals()
Sort()(Sort<T>)	IComparer (IComparer<T>)
BinarySearch() (BinarySearch<T>)	IComparer (IComparer<T>)

Table 30-5. *ArrayList Interface (List<T>) and Virtual Method Uses*

Function	Uses
IndexOf()	Equals()
LastIndexOf()	Equals()
Contains()	Equals()
Sort()	IComparer (IComparer<T> or Comparison<T>)
BinarySearch()	IComparer (IComparer<T>)

Table 30-6. *Hashtable Interface (Dictionary<T>) and Virtual Method Uses*

Function	Uses
Hashtable()	IHashCodeProvider, IComparer (IComparer<T>) (optional)
Contains()	GetHashCode(), Equals()
Item	GetHashCode(), Equals()

Table 30-7. *SortedList Interface (SortedList<T>) and Virtual Method Uses*

Function	Uses
SortedList()	IComparer (IComparer<T>)
Contains()	IComparer (IComparer<T>)
ContainsKey()	IComparer (IComparer<T>)
ContainsValue()	Equals()
IndexOfKey()	IComparer (IComparer<T>)
IndexOfValue()	Equals()
Item	IComparer (IComparer<T>)

Choosing Generics vs. Nongeneric Collections

You should use generic collections as often as possible, as opposed to using nongeneric collections. The generic collections are much faster (up to an order of magnitude) when working with value types and are still significantly faster when working with reference types. The generic collections also provide compile-time type-safety that isn't available with the nongeneric collections; further, the designers incorporated what worked best in version 1 of the .NET Framework libraries into the design and implementation of generic collections.

Generics are CLS-compliant in .NET 2.0, which means the range of languages that will be able to consume, extend, and produce generic classes will be substantial, so shying away from generic collections in the interests of language interoperability is unwarranted.

CHAPTER 31

Threading and Asynchronous Operations

Modern computer operating systems allow a program to have multiple threads of execution at one time. At least, they allow the appearance of having multiple things going on at the same time.

It's often useful to take advantage of this feature with a programming language and allow several operations to take place in parallel. You can use this to prevent a program's user interface from becoming unresponsive while a time-consuming task is being performed, or you can use it to execute some other task while waiting for a blocking operation (an I/O, for example) to complete.

The CLR provides two ways to perform such operations: through threading and through asynchronous call mechanisms.

Note that this is a big topic, and the material in this chapter is a starting point for real-world applications. Large-scale multithreading is a complex and demanding topic and is covered in greater depth in books such as *Maximizing .NET Performance* (Apress, 2003) by Nick Wienholt and *Expert .NET 1.1 Programming* (Apress, 2004) by Simon Robinson.

Data Protection and Synchronization

Performing more than one operation at once provides a valuable facility to a program, but it also increases the complexity of the programming task.

A Slightly Broken Example

Consider the following code:

```
using System;
class Val
{
    int number = 1;

    public void Bump()
    {
        int temp = number;
        number = temp + 2;
    }
```

```
    public override string ToString()
    {
        return(number.ToString());
    }

    public void DoBump()
    {
        for (int i = 0; i < 5; i++)
        {
            Bump();
            Console.WriteLine("number = {0}", number);
        }
    }
}

class Test
{
    public static void Main()
    {
        Val v = new Val();

        v.DoBump();
    }
}
```

In this example, the `Val` class holds a number and has a way to add 2 to it. When this program runs, it generates the following output:

```
number = 3
number = 5
number = 7
number = 9
number = 11
```

While that program is being executed, the operating system may be performing other tasks simultaneously. The code can be interrupted at any spot in the code,[1] but after the interruption, everything will be in the same state as before, and you have no way of knowing that the interruption took place.

1. Not quite *any* spot; the situations where it won't be interrupted are covered later.

You can modify the program to use some threads:

```
using System;
using System.Threading;
class Val
{
    int number = 1;

    public void Bump()
    {
        int temp = number;
        number = temp + 2;
    }

    public override string ToString()
    {
        return(number.ToString());
    }

    public void DoBump()
    {
        for (int i = 0; i < 5; i++)
        {
            Bump();
            Console.WriteLine("number = {0}", number);
        }
    }
}

class Test
{
    public static void Main()
    {
        Val v = new Val();

        for (int threadNum = 0; threadNum < 5; threadNum++)
        {
            Thread thread = new Thread(new ThreadStart(v.DoBump));
            thread.Start();
        }
    }
}
```

In this code, a `ThreadStart` delegate is created that refers to the function the thread should execute. When this program runs, it generates the following output:

```
number = 3
number = 5
number = 7
number = 9
number = 11
number = 13
number = 15
number = 17
number = 19
number = 21
number = 23
number = 25
number = 27
number = 29
number = 31
number = 33
number = 35
number = 37
number = 39
number = 41
number = 43
number = 45
number = 47
number = 49
number = 51
```

Can you find the error in the output? No?

This example illustrates one of the common problems with writing multithreaded programs. The example has a latent error that might show up in some situations, but it doesn't show up when the example runs under normal conditions. Bugs such as these are some of the worst to find, and they usually show up only under stressful conditions (such as at a customer's site).

You can change the code a bit to simulate an interruption by the operating system:

```
public void Bump()
{
    int temp = number;
    Thread.Sleep(1);
    number = temp + 2;
}
```

This small change leads to the following output:

```
number = 3
number = 3
number = 3
number = 3
number = 3
number = 5
number = 5
number = 5
number = 5
number = 5
number = 7
number = 7
number = 7
number = 7
number = 7
number = 9
number = 9
number = 9
number = 9
number = 9
number = 11
number = 11
number = 11
number = 11
number = 11
```

This isn't exactly the desired result.

The call to `Thread.Sleep()` will cause the current thread to sleep for one millisecond, before it has saved the bumped value. When this happens, another thread will come in and also fetch the current value.

The underlying bug in the code is that you have no protection against this situation happening. Unfortunately, it's rare enough that it's hard to find. Creating multithreaded applications is one area where good design techniques are important.

Protection Techniques

You can use several techniques to prevent problems. Code that's written to keep this in mind is known as *thread-safe*.

In general, most code isn't thread-safe, because there's usually a performance penalty in writing thread-safe code.

Don't Share Data

One of the best techniques to prevent such problems is to not share data in the first place. It's often possible to architect an application so that each thread has its own data to deal with. An application that fetches data from several Web sites simultaneously can create a separate object for each thread.

This is obviously the best option, as it imposes no performance penalty and doesn't clutter the code. It requires some care, however, since modifying the code may introduce the errors you tried to carefully avoid. For example, a programmer who doesn't know that a class uses threading might add shared data.

Immutable Objects

Immutable objects are thread-safe by definition. Multiple threads can safely read a piece of data simultaneously, and it's only when modifying operations are taking place that precautions need to be taken in a multithreaded scenario. Because immutable objects don't allow modifying operations, you don't need any other thread-safety measures.

The `string` type is a great example of achieving thread-safety through immutability. Every modifying operation on a `string`, such as `ToUpper()`, returns a new `string` instance. If one thread is completing an enumeration of the characters of a string at the same time as another thread calls `ToUpper()`, the thread conducting the enumeration is unaffected because the new uppercase string is an entirely separate object that isn't physically related to the original `string`.

Immutability does place a higher design and implementation burden on a type. The methods of the type must be designed so it's apparent to the users of the type that a modifying operation returns a new instance rather than modifying the instance it was called, and it's generally wise to provide a mutable equivalent of the immutable type to support high-performance modification operations. For `string`, `StringBuilder` is the equivalent mutable type.

Exclusion Primitives

The `System.Threading` namespace contains a number of useful classes for preventing the problems in the earlier example. The most commonly used one is the `Monitor` class. You can modify the slightly broken example by surrounding the problem region of code with exclusion primitives:

```
public void Bump()
{
    Monitor.Enter(this);
    int temp = number;
    Thread.Sleep(1);
    number = temp + 2;
    Monitor.Exit(this);
}
```

The call to `Monitor.Enter()` passes in the `this` reference for this object. The monitor's job is to make sure that if a thread has called `Monitor.Enter()` with a specific value, any other call to `Monitor.Enter()` with the same value will block until the first thread has called `Monitor.Exit()`. When the first thread calls `Thread.Sleep()`, the second thread will call `Monitor.Enter()` and pass the same object as the first thread did, and therefore the second thread will block.

■ **Note** Those of you who've done Win32 programming may be familiar with using `EnterCriticalSection()` and `LeaveCriticalSection()` to block access. Unlike the Win32 functions, the `Monitor` functions lock on a specific object.

The implementation of `Bump()` has a slight problem. If an exception was thrown in the block that's protected, `Monitor.Exit()` will never be called, which is bad. To make sure `Monitor.Exit()` is always called, the calls need to be wrapped in a `try-finally`. This is important enough that C# provides a special statement to do just that.

The lock Statement

The `lock` statement is simply a thin wrapper around calls to `Monitor.Enter()` and `Monitor.Exit()`. The following code:

```
object lockObj = new object();
lock(lockObj)
{
    // statements
}
```

translates to the following:

```
object lockObj = new object();
try
{
    System.Threading.Monitor.Enter(lockObj);
    // statements
}
finally
{
    System.Threading.Monitor.Exit(lockObj);
}
```

The object that's used in the `lock` statement reflects the granularity at which the lock should be obtained. If the data to be protected is instance data, it's typical to create a private member variable and use this to prevent concurrent access.

If the data to be protected is a static data item, it will need to be locked using a unique static reference object. You do this simply by adding a static field of type `object` to the class:

```
static object staticLock = new object();
```

This object is then used in the `lock` statement.

For types that will be widely distributed, creating a private member variable to use in `lock` statements is preferable to taking out a `lock` on `this`. Code anywhere inside an application domain can also take out a `lock` on the `this` object reference, which can cause contention and deadlocks. Using a private member variable for locking is extremely simple to implement:

```
public class WidleyDistributedType
{
 private object wdtLock = new object();

 public void ThreadSafeMethod()
 {
  lock(wdtLock) {/*code here*/}
 }
}
```

Synchronized Methods

An alternate to using the `lock` statement is to mark the entire method as synchronized, which has the same effect as enclosing the entire method in `lock(this)`. To do this, mark the method with the following attribute:

```
[MethodImpl(MethodImplOptions.Synchronized)]
```

You can find this attribute in the `System.Runtime.CompilerServices` namespace.

In general, you should use `lock` over the attribute for two reasons. First, the performance is better because the region in which a lock exists is often a subset of the whole method. Second, it's easy to miss the attribute (or forget what it does) when reading code; using `lock` is more explicit.

Interlocked Operations

Many processors support some instructions that can't be interrupted. These are useful when dealing with threads, as no locking is required to use them. In the CLR, these operations are encapsulated in the `Interlocked` class in the `System.Threading` namespace. This class exposes the `Increment()`, `Decrement()`, `Exchange()`, and `CompareExchange()` methods, which can be used on `int` or `long` data types.

You could rewrite the problem example using these instructions:

```
public void Bump()
{
    Interlocked.Increment(ref number);
    Interlocked.Increment(ref number);
}
```

The runtime guarantees the increment operations won't be interrupted. If `Interlocked` works for an application, it can provide a nice performance boost, as it avoids the overhead of locking.

Mutexes and Semaphores

The exclusion primitives discussed so far are all application domain–specific locking techniques. A `Monitor` object in one application domain is totally unaffected by a `Monitor` object is another application domain or process, even if they have the same name. When protecting against concurrent access to resources that are specific to application domains such as C# objects, this is the behavior that's desired. However, at times, you'll need to protect globally available

resource such as files, ports, and hardware devices. In these cases, you'll need systemwide exclusion primitives. This is where the `Mutex` and `Semaphore` types come in.

`Mutex` acts in the same way as `Monitor` but is unique across all processes on a machine. Acquiring a lock on a `Mutex` in one process blocks all other processes from acquiring a lock on the same mutex. The global nature of `Mutex` makes it much more expensive to acquire than a `Monitor` lock—about two orders of magnitude slower. `Mutex` has a different syntax for acquiring a lock and doesn't have a C# language helper equivalent to a `Monitor`'s `lock` statement. The following code shows a `Mutex` in use:

```
//create a Mutex called "PORT_1234_PROTECT".  The mutex is not initially
//owned by this thread
Mutex portProtector = new Mutex(false, "PORT_1234_PROTECT");
try
{
 //acquire the mutex
 portProtector.WaitOne();
 //operations on port that need to be protected
}
finally
{
 portProtector.ReleaseMutex();
}
```

.NET 2.0 introduces a new exclusion primitive called a `Semaphore`. A `Semaphore` is similar to a `Mutex`, but where a `Mutex` protects access only to a single resource, a `Semaphore` protects access to multiple instances of a similar resource. Consider a call center: a customer's call can be routed to any available call center staffer, but as there are usually more callers than staff and a staffer can take only a single call at a time, access to the staffer needs to be locked until any of the staffers become available. A pool of objects shares the same requirement—any object from the pool is as good as the next from a client's perspective, but because the number of objects in the pool is finite, a locking mechanism needs to be put in place to prevent two clients from acquiring the same pooled object.

Using a `Semaphore` is almost identical to `Mutex`. The only difference for a programming model perspective is that a maximum count needs to be set when the `Semaphore` is created:

```
//create a Semaphore called "PORT_12xx_PROTECT". The thread has an initial count
//of 0 on the semaphore. The semaphore has a maximum count of 6
Semaphore mutiplePortProtector = new Semaphore(0, 6, "PORT_12xx_PROTECT");
try
{
 mutiplePortProtector.WaitOne();
 //operations on port that need to be protected
}
finally
{
 mutiplePortProtector.Release();
}
```

Access Reordering and Volatile

To avoid the overhead of synchronization, some programmers will build their own optimization primitives. In C#, however, some surprising subtleties exist in what the language and runtime guarantee with respect to instruction ordering, especially to those who are familiar with the x86 architecture, which doesn't typically perform these operations.

This topic is complex, but it isn't necessary to fully understand it if you stick to the synchronization methods discussed earlier in this chapter.

To illustrate this, consider the following example:

```
using System;
using System.Threading;

class Problem
{
    int x;
    int y;
    int curx;
    int cury;

    public Problem()
    {
        x = 0;
        y = 0;
    }

    public void Process1()
    {
        x = 1;
        cury = y;
    }

    public void Process2()
    {
        y = 1;
        curx = x;
    }

    public void TestCurrent()
    {
        Console.WriteLine("curx, cury: {0} {1}", curx, cury);
    }
}
```

```
class Test
{
    public static void Main()
    {
        Problem p = new Problem();

        Thread t1 = new Thread(new ThreadStart(p.Process1));
        Thread t2 = new Thread(new ThreadStart(p.Process2));
        t1.Start();
        t2.Start();

        t1.Join();
        t2.Join();

        p.TestCurrent();
    }
}
```

In this example, two threads are started: one that calls `p.Process1()` and another that calls `p.Process2()`. Figure 31-1 shows this process.

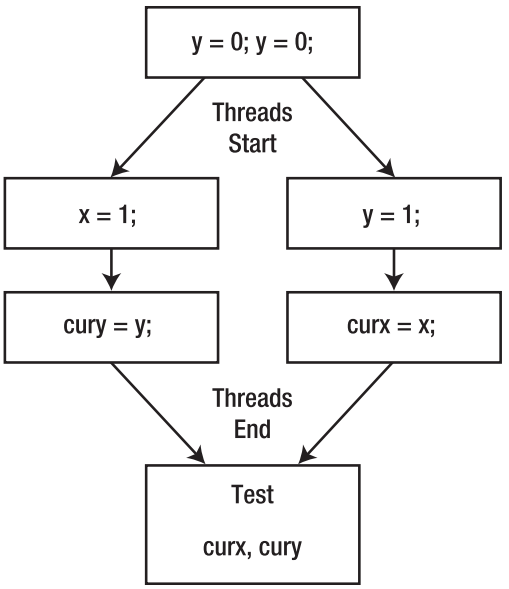

Figure 31-1. *Reordering example*

What possible values can be printed for curx and cury? It's not surprising that the following are two possible values:

```
curx, cury: 1 0
curx, cury: 0 1
```

This makes sense from the serial nature of the code in Process1() and Process2(); either function can complete before the other one starts.

The following output is a bit less obvious:

```
curx, cury: 1 1
```

This is a possibility because one of the threads could be interrupted after the first instruction and the other thread could run.

The point of the example, however, is that there's a fourth possible output:

```
curx, cury: 0 0
```

This happens because of one of those things that's assumed to be true but isn't really always true. The common assumption when looking at the code in Process1() is that the lines always execute in the order in which they're written. Surprisingly, that isn't true; the following are a few cases where the instructions might execute out of order:

- The compiler could choose to reorder the statements, as there's no way for the compiler to know this isn't safe.
- The JIT could decide to load the values for both x and y into registers before executing either line of code.
- The processor could reorder the execution of the instructions to be faster.[2]
- On a multiprocessor system, the values might not be synchronized in global memory.

What's needed to address this is a way to annotate a field so that such optimizations are inhibited. C# does this with the volatile keyword.

When a field is marked as volatile, reordering of instructions is inhibited, so the following is true:

- A write can't be moved forward across a volatile write.
- A read can't be moved backward across a volatile read.

2. x86 processors don't do this, but there are other processors—including Intel's IA64 architecture—where reordering is common.

In the example, if curx and cury are marked volatile, the code in Process1() and Process2() can't be reordered:

```
public void Process1()
{
    x = 1;
    cury = y;
}
```

Since cury is now volatile, the write to x can't be moved after the write to cury.

In addition to precluding such reordering, volatile also means that the JIT can't keep the variable in the register and that the variable must be stored in global memory on a multiprocessor system.

So what's volatile good for, when you already have ways of doing synchronization?

Using volatile

You can use volatile to implement a thread-safe version of a singleton class. The traditional implementation uses lock:

```
using System;
class Singleton
{
    static object sync = new object();
    static Singleton singleton = null;

    private Singleton()
    {
    }

    public static Singleton GetSingleton()
    {
        lock(sync)
        {
            if (singleton == null)
                singleton = new Singleton();

            return(singleton);
        }
    }
}
```

This works fine, but it's pretty wasteful; the synchronization is really needed only the first time the function is called. In this case, the lock needs to be on a static variable because the function is a static function.

With volatile, you can write a nicer version:

```
using System;

class Singleton
{
    static object sync = new object();

    static volatile Singleton singleton = null;

    private Singleton()
    {
    }

    public static Singleton GetSingleton()
    {
        if (singleton == null)
        {
            lock(sync)
            {
                //check for null again in case another thread has assigned
                //a value between the previous check and the lock statement
                if (singleton == null)
                    singleton = new Singleton();
            }
        }

        return(singleton);
    }
}
```

This version has much better performance since the synchronization is required only if the object hasn't been created.

Threads

The previous example has shown a bit about threads, but we have to cover a few more details. When a Thread instance is created, a delegate to the function that the thread should run is created and passed to the constructor. Since a delegate can refer to a member function and a specific instance, there's no need to pass anything else to the thread.

You can use the thread instance to control the priority of the thread, set the name of the thread, or perform other thread operations, so it's necessary to save that thread instance if such operations will be performed later. A thread can get its own thread instance through the Thread.CurrentThread property.

Joining

After a thread has been created to perform a task, such as doing a computation-intensive calculation, it's sometimes necessary to wait for that thread to complete. The following example illustrates this:

```
using System;
using System.Threading;

class ThreadSleeper
{
    int seconds;

    private ThreadSleeper(int seconds)
    {
        this.seconds = seconds;
    }

    public void Nap()
    {
        Console.WriteLine("Napping {0} seconds", seconds);
        Thread.Sleep(seconds * 1000);
    }

    public static Thread DoSleep(int seconds)
    {
        ThreadSleeper ts = new ThreadSleeper(seconds);
        Thread thread = new Thread(new ThreadStart(ts.Nap));
        thread.Start();
        return(thread);
    }
}

class Test
{
    public static void Main()
    {
        Thread thread = ThreadSleeper.DoSleep(5);

        Console.WriteLine("Waiting for thread to join");
        thread.Join();
        Console.WriteLine("Thread Joined");
    }
}
```

The `ThreadSleeper.Nap()` function simulates an operation that takes a while to perform. `ThreadSleeper.DoSleep()` creates an instance of a `ThreadSleeper`, executes the `Nap()` function, and then returns the thread instance to the main program. The main program then calls `Join()` on that thread to wait for it to complete.

Using `Join()` works well when waiting for a single thread, but if there's more than one active thread, a call to `Join()` must be made for each, which is a bit unwieldy.

A nicer solution is to use one of the utility classes.

Waiting with WaitHandle

The `WaitHandle` abstract class provides a simple way to wait for an event to occur.[3] In addition to waiting for a single event to occur, it can be used to wait for more than one event and return when one or all of them occur. The `AutoResetEvent` and `ManualResetEvent` classes derive from `WaitHandle`. The `AutoResetEvent` will release only a single thread when the `Set()` function is called and will then reset. The `ManualResetEvent` may release many threads from a single call to `Set()` and must be cleared by calling `Reset()`.

You can modify the previous example to use an `AutoResetEvent` to signal when an event is complete and to wait for more than one thread to complete:

```
using System;
using System.Threading;

class ThreadSleeper
{
    int seconds;
    AutoResetEvent napDone = new AutoResetEvent(false);

    private ThreadSleeper(int seconds)
    {
        this.seconds = seconds;
    }

    public void Nap()
    {
        Console.WriteLine("Napping {0} seconds", seconds);
        Thread.Sleep(seconds * 1000);
        Console.WriteLine("{0} second nap finished", seconds);
        napDone.Set();
    }
```

3. Note that this is an event in the general sense, not in the "C# event" sense.

```
    public static WaitHandle DoSleep(int seconds)
    {
        ThreadSleeper ts = new ThreadSleeper(seconds);
        Thread thread = new Thread(new ThreadStart(ts.Nap));
        thread.Start();
        return(ts.napDone);
    }
}

class Test
{
    public static void Main()
    {
        WaitHandle[] waits = new WaitHandle[2];
        waits[0] = ThreadSleeper.DoSleep(8);
        waits[1] = ThreadSleeper.DoSleep(4);

        Console.WriteLine("Waiting for threads to finish");
        WaitHandle.WaitAll(waits);
        Console.WriteLine("Threads finished");
    }
}
```

The output is as follows:

```
Waiting for threads to finish
Napping 8 seconds
Napping 4 seconds
4 second nap finished
8 second nap finished
Threads finished
```

Instead of returning a `Thread`, the `DoSleep()` function now returns a `WaitHandle` that will be set when the thread has completed. An array of `WaitHandle` is created and then is passed to `WaitHandle.WaitAll()` to wait for all the events to be set.

Asynchronous Calls

When using threads, the programmer is responsible for taking care of all the details of execution and determining how to transfer data from the caller to the thread and then back from the thread. This normally involves creating a class to encapsulate the data to be exchanged, which is a fair bit of extra work.

Writing this class isn't difficult, but it's a bit of a pain to do if all that's needed is a single asynchronous call. Luckily, the runtime and compiler provide a way to get asynchronous execution without a separate class.

The runtime will handle the details of managing the thread on which the function will be called (using a thread pool) and provide an easy mechanism for exchanging data. Nicer still, the runtime will allow *any* function to be called with this mechanism; it doesn't have to be designed to be asynchronous to call it asynchronously. This can be a nice way to start an operation and then continue with the main code.

It all happens through a little magic in delegates.

To set up an asynchronous call, the first step is to define a delegate that matches the function to be called. For example, if the function is as follows:

```
Console.WriteLine(string s);
```

the delegate would be like this:

```
delegate void FuncToCall(string s);
```

If you place this delegate in a class and compile it, you can view it using the ILDASM utility. An `Invoke()` member takes a string, which invokes a delegate, and then there are two strange-looking functions:

```
public IAsyncResult BeginInvoke(string s, System.AsyncCallback callback, object o);
public void EndInvoke(IAsyncResult);
```

These functions are generated by the compiler to make doing asynchronous calls easier and are defined based upon the parameters and return type of the delegate, as detailed in Table 31-1.

Table 31-1. *Parameter Passing with Asynchronous Methods*

Data	Mechanism
Value parameters	Passed to `BeginInvoke()`
Ref parameters	Passed as `ref` to `BeginInvoke()` and `EndInvoke()`
Out parameters	Passed as `out` to `EndInvoke()`
Return value	Returned from `EndInvoke()`

In addition to the parameters defined for the delegate, `BeginInvoke()` also takes an optional callback to call when the function call has completed and an object that can be used by the caller to pass some state information. `BeginInvoke()` returns an `IAsyncResult` that's passed to `EndInvoke()`.

A Simple Example

The following example shows a simple async call:

```
using System;
public class AsyncCaller
{
    // Declare a delegate that will match Console.WriteLine("string");
    delegate void FuncToCall(string s);

    public void CallWriteLine(string s)
    {
            // delegate points to function to call
            // start the async call
            // wait for completion
        FuncToCall func = new FuncToCall(Console.WriteLine);
        IAsyncResult iar = func.BeginInvoke(s, null, null);
        func.EndInvoke(iar);
    }
}

class Test
{
    public static void Main()
    {
        AsyncCaller ac = new AsyncCaller();
        ac.CallWriteLine("Hello");
    }
}
```

The `CallWriteLine()` function takes a `string` parameter, creates a delegate to `Console.WriteLine()`, and then calls `BeginInvoke()` and `EndInvoke()` to call the function asynchronously and wait for it to complete.

That's not terribly exciting. Let's modify the example to use a callback function:

```
using System;

public class AsyncCaller
{
    // Declare a delegate that will match Console.WriteLine("string");
    delegate void FuncToCall(string s);

    public void WriteLineCallback(IAsyncResult iar)
    {
```

```
        Console.WriteLine("In WriteLineCallback");
        FuncToCall func = (FuncToCall) iar.AsyncState;
        func.EndInvoke(iar);
    }

    public void CallWriteLineWithCallback(string s)
    {
        FuncToCall func = new FuncToCall(Console.WriteLine);
        func.BeginInvoke(s,
                                    new AsyncCallback(WriteLineCallback),
                                    func);
    }
}
class Test
{
    public static void Main()
    {
        AsyncCaller ac = new AsyncCaller();

        ac.CallWriteLineWithCallback("Hello There");

        System.Threading.Thread.Sleep(1000);
    }
}
```

The `CallWriteLineWithCallback()` function calls `BeginInvoke()`, passing a callback function and the delegate. The callback routine takes the callback function passed in the state object and calls `EndInvoke()`.

Because the call to `CallWriteLineWithCallback()` returns immediately, the `Main()` function sleeps for a second so the asynchronous call can complete before the program exits.

Return Values

This example calls `Math.Sin()` asynchronously:

```
using System;
using System.Threading;

public class AsyncCaller
{
    public delegate double MathFunctionToCall(double arg);

    public void MathCallback(IAsyncResult iar)
    {
        MathFunctionToCall mc = (MathFunctionToCall) iar.AsyncState;
        double result = mc.EndInvoke(iar);
        Console.WriteLine("Function value = {0}", result);
    }
```

```
    public void CallMathCallback(MathFunctionToCall mathFunc,
                                                 double start,
                                                 double end,
                                                 double increment)
    {
        AsyncCallback cb = new AsyncCallback(MathCallback);

        while (start < end)
        {
            Console.WriteLine("BeginInvoke: {0}", start);
            mathFunc.BeginInvoke(start, cb, mathFunc);
            start += increment;
        }
    }
}

class Test
{
    public static void Main()
    {
        AsyncCaller ac = new AsyncCaller();

        ac.CallMathCallback(
             new AsyncCaller.MathFunctionToCall(Math.Sin), 0.0, 1.0, 0.2);
        Thread.Sleep(2000);
    }
}
```

This generates the following output:

```
BeginInvoke: 0
BeginInvoke: 0.2
BeginInvoke: 0.4
BeginInvoke: 0.6
BeginInvoke: 0.8
Function value = 0
Function value = 0.198669330795061
Function value = 0.389418342308651
Function value = 0.564642473395035
Function value = 0.717356090899523
```

This time, the call to `EndInvoke()` in the callback returns the result of the function, which is then written out. Note that `BeginInvoke()` gets called before any of the calls to `Math.Sin()` occur.

Because the example doesn't contain any synchronization, the call to `Thread.Sleep` is required to make sure the callbacks execute before main finishes. It's worth noting that the call to `Thread.Sleep` is only an example, and you should use the proper synchronization in production code.

Waiting for Completion

It's possible to wait for several asynchronous calls to finish, using `WaitHandle` as in the threads section. The `IAsyncResult` returned from `BeginInvoke()` has an `AsyncWaitHandle` member that can be used to know when the asynchronous call completes. Here's a modification to the previous example:

```
using System;
using System.Threading;

public class AsyncCaller
{
    public delegate double MathFunctionToCall(double arg);

    public void MathCallback(IAsyncResult iar)
    {
        MathFunctionToCall mc = (MathFunctionToCall) iar.AsyncState;
        double result = mc.EndInvoke(iar);
        Console.WriteLine("Function value = {0}", result);
    }

    WaitHandle DoInvoke(MathFunctionToCall mathFunc, double value)
    {
        AsyncCallback cb = new AsyncCallback(MathCallback);

        IAsyncResult asyncResult =
            mathFunc.BeginInvoke(value, cb, mathFunc);
        return(asyncResult.AsyncWaitHandle);
    }

    public void CallMathCallback(MathFunctionToCall mathFunc)
    {
        WaitHandle[] waitArray = new WaitHandle[4];

        Console.WriteLine("Begin Invoke");
        waitArray[0] = DoInvoke(mathFunc, 0.1);
        waitArray[1] = DoInvoke(mathFunc, 0.5);
        waitArray[2] = DoInvoke(mathFunc, 1.0);
        waitArray[3] = DoInvoke(mathFunc, 3.14159);
        Console.WriteLine("Begin Invoke Done");

        Console.WriteLine("Waiting for completion");
        WaitHandle.WaitAll(waitArray, 10000, false);
        Console.WriteLine("Completion achieved");
    }
}
```

```
public class Test
{
    public static double DoCalculation(double value)
    {
        Console.WriteLine("DoCalculation: {0}", value);
        Thread.Sleep(250);
        return(Math.Cos(value));
    }

    public static void Main()
    {
        AsyncCaller ac = new AsyncCaller();

        ac.CallMathCallback(new AsyncCaller.MathFunctionToCall(DoCalculation));
        //Thread.Sleep(500);        // no longer needed
    }
}
```

The `DoInvoke()` function returns the `WaitHandle` for a specific call, and `CallMathCallback()` waits for all the calls to complete and then returns. Because of the wait, the sleep call in `Main` is no longer needed.

This example generates the following output:

```
Begin Invoke
Begin Invoke Done
Waiting for completion
DoCalculation: 0.1
Function value = 0.995004165278026
DoCalculation: 0.5
Function value = 0.877582561890373
DoCalculation: 1
Function value = 0.54030230586814
DoCalculation: 3.14159
Completion achieved
```

The return value for the last calculation is missing.

This illustrates a problem with using the `WaitHandle` that's provided in the `IAsyncResult`. The `WaitHandle` is set when `EndInvoke()` is called but before the callback routine completes. In this example, it's obvious that something's wrong, but in a real program, this could result in a really nasty race condition, where some results are dropped. This means that using the provided `WaitHandle` is safe only if there isn't any processing done after `EndInvoke()`.

The way to deal with this problem is to ignore the provided `WaitHandle` and add a `WaitHandle` that's called at the end of the callback function:

```
using System;
using System.Threading;

public class AsyncCallTracker
{
    Delegate function;
    AutoResetEvent doneEvent;

    public AutoResetEvent DoneEvent
    {
        get
        {
            return(doneEvent);
        }
    }

    public Delegate Function
    {
        get
        {
            return(function);
        }
    }

    public AsyncCallTracker(Delegate function)
    {
        this.function = function;
        doneEvent = new AutoResetEvent(false);
    }
}

public class AsyncCaller
{
    public delegate double MathFunctionToCall(double arg);

    public void MathCallback(IAsyncResult iar)
    {
        AsyncCallTracker callTracker = (AsyncCallTracker) iar.AsyncState;
        MathFunctionToCall func = (MathFunctionToCall) callTracker.Function;
        double result = func.EndInvoke(iar);
        Console.WriteLine("Function value = {0}", result);
        callTracker.DoneEvent.Set();
    }
```

```
    WaitHandle DoInvoke(MathFunctionToCall mathFunc, double value)
    {
        AsyncCallTracker callTracker = new AsyncCallTracker(mathFunc);

        AsyncCallback cb = new AsyncCallback(MathCallback);
        IAsyncResult asyncResult = mathFunc.BeginInvoke(value, cb, callTracker);
        return(callTracker.DoneEvent);
    }

    public void CallMathCallback(MathFunctionToCall mathFunc)
    {
        WaitHandle[] waitArray = new WaitHandle[4];

        Console.WriteLine("Begin Invoke");
        waitArray[0] = DoInvoke(mathFunc, 0.1);
        waitArray[1] = DoInvoke(mathFunc, 0.5);
        waitArray[2] = DoInvoke(mathFunc, 1.0);
        waitArray[3] = DoInvoke(mathFunc, 3.14159);
        Console.WriteLine("Begin Invoke Done");

        Console.WriteLine("Waiting for completion");
        WaitHandle.WaitAll(waitArray, 10000, false);
        Console.WriteLine("Completion achieved");
    }
}

public class Test
{
    public static double DoCalculation(double value)
    {
        Console.WriteLine("DoCalculation: {0}", value);
        Thread.Sleep(250);
        return(Math.Cos(value));
    }

    public static void Main()
    {
        AsyncCaller ac = new AsyncCaller();

        ac.CallMathCallback(new AsyncCaller.MathFunctionToCall(DoCalculation));
    }
}
```

It's now necessary to pass both the delegate and an associated `AutoResetEvent` to the callback function, so these are encapsulated in the `AsyncCallTracker` class. The `AutoResetEvent` is returned from `DoInvoke()`, and this event isn't set until the last line of the callback, so there are no longer any race conditions.

Classes That Support Asynchronous Calls Directly

Some framework classes provide explicit support for asynchronous calls, which allows them to have full control over how asynchronous calls are processed. The `HttpWebRequest` class, for example, provides `BeginGetResponse()` and `EndGetResponse()` functions, so creating a delegate isn't required. Windows Forms also has its own built-in threading support to allow a somewhat simplified model to work around the fact that the underlying Windows API isn't thread-safe. `Control.BeginInvoke` (inherited by many classes in `Windows.Forms`) and `Control.IsInvokeRequired` are the workhorses of the Windows Forms threading functionality.

All the framework classes that provide such support adhere to the same pattern as the do-it-yourself approach and are used in the same manner.

Design Guidelines

Both threading and asynchronous calls provide a way to have more than one path of execution happen at once. In most situations, you can use either method.

Asynchronous calls are best for situations where you're doing one or two asynchronous calls and you don't want the hassle of setting up a separate thread or dealing with data transfer. The system uses a thread pool to implement asynchronous calls (see the "Thread Pools" sidebar), and you have no way to control how many threads it assigns to processing asynchronous calls or anything else about the thread pool. Because of this, asynchronous calls aren't suited for more than a few active calls at once.

Threads allow more flexibility than asynchronous calls but often require more design and implementation work, especially if a thread pool needs to be implemented. It's also more work to transfer data around, and synchronization details may require more thought.

There's also a readability issue. Thread-based code is often easier to understand (though it's probably more likely to harbor hard-to-find problems), and it's a more familiar idiom.

THREAD POOLS

A program with simple threading will often create a separate thread for every operation. Not only is this wasteful, as threads are being created and destroyed continuously, but it scales poorly, because having 1,000 threads isn't good design (if it works at all).

A way to get around this is to use a thread pool. Rather than having a thread dedicated to a specific operation, a small number of worker threads are created. Incoming operations are assigned to a specific worker thread (or queued if all threads are busy), so when a thread is done with an operation, it waits for another one to perform.

Advanced thread pools manage the number of worker threads on the fly based on the queue length and other factors.

CHAPTER 32

Execution-Time Code Generation

If you come from a C++ background, you may have a "compile-time" view of the world. Because a C++ compiler does all code generation when the code is compiled, C++ programs are static systems that are fully known at compile time.

The CLR provides a new way of doing things. The compile-time world still exists, but it's also possible to build dynamic systems where new code is added by loading assemblies or even by writing custom code on the fly.

Loading Assemblies

In the .NET CLR, it's possible to load an assembly from disk and to create instances of classes from that assembly. To demonstrate this, this chapter shows how to build a simple logging facility that can be extended by the customer at runtime to send informational messages elsewhere.

The first step is to define the standard part of the facility:

```
// file=LogDriver.cs
// compile with: csc /target:library LogDriver.cs
using System;
using System.Collections;

public interface ILogger
{
    void Log(string message);
}

public class LogDriver
{
    ArrayList loggers = new ArrayList();

    public LogDriver()
    {
    }
```

```
    public void AddLogger(ILogger logger)
    {
        loggers.Add(logger);
    }

    public void Log(string message)
    {
        foreach (ILogger logger in loggers)
        {
            logger.Log(message);
        }
    }
}

public class LogConsole: ILogger
{
    public void Log(string message)
    {
        Console.WriteLine(message);
    }
}
```

In this step, you define the `ILogger` interface that your loggers will implement and the `LogDriver` class that calls all the registered loggers whenever the `Log()` function is called. Also, a `LogConsole` implementation logs messages to the console. This file is compiled to an assembly named `LogDriver.dll`.

In addition to this file, a small class exercises the loggers:

```
using System;

class Test
{
    public static void Main()
    {
        LogDriver logDriver = new LogDriver();

        logDriver.AddLogger(new LogConsole());

        logDriver.Log("Log start: " + DateTime.Now.ToString());

        for (int i = 0; i < 5; i++)
        {
            logDriver.Log("Operation: " + i.ToString());
        }

        logDriver.Log("Log end: " + DateTime.Now.ToString());
    }
}
```

This code merely creates a `LogDriver`, adds a `LogConsole` to the list of loggers, and does some logging.

Making It Dynamic

It's now time to add some dynamic ability to the system. You'll need a mechanism so the `LogDriver` class can discover there's a new assembly that contains an additional logger. To keep the sample simple, the code will look for assemblies named `LogAddIn*.dll`.

The first step is to come up with another implementation of `ILogger`. The `LogAddInToFile` class logs messages to `logger.log`:

```
// file=LogAddInToFile.cs
// compile with: csc /r:..\logdriver.dll /target:library logaddintofile.cs
using System;
using System.Collections;
using System.IO;

public class LogAddInToFile: ILogger
{
    StreamWriter streamWriter;

    public LogAddInToFile()
    {
        streamWriter = File.CreateText(@"logger.log");
        streamWriter.AutoFlush = true;
    }

    public void Log(string message)
    {
        streamWriter.WriteLine(message);
    }
}
```

This class doesn't require much explanation. Next, you need to add the code to load the assembly to the `LogDriver` class:

```
    void ScanDirectoryForLoggers()
    {
        DirectoryInfo dir = new DirectoryInfo(@".");
        foreach (FileInfo f in dir.GetFiles(@"LogAddIn*.dll"))
        {
            ScanAssemblyForLoggers(f.FullName);
        }
    }
    void ScanAssemblyForLoggers(string filename)
    {
        Assembly a = Assembly.LoadFrom(filename);
```

```
        foreach (Type t in a.GetTypes())
        {
            if (t.GetInterface("ILogger") != null)
            {
                ILogger iLogger = (ILogger) Activator.CreateInstance(t);
                loggers.Add(iLogger);
            }
        }
    }
}
```

The `ScanDirectoryForLoggers()` function looks in the current directory for any files that match your specification. When one of the files is found, `ScanAssemblyForLoggers()` is called. This function loads the assembly and then iterates through each of the types contained in the assembly. If the type implements the `ILogger` interface, then an instance of the type is created using `Activator.CreateInstance()`, the instance is cast to the interface, and the interface is added to the list of loggers.

If you desire an even more dynamic implementation, you could use a `FileChangeWatcher` object to watch a specific directory, and you could then load any assemblies copied to that directory.

A few caveats exist with regard to loading assemblies from a disk. First, the runtime locks assemblies when they're loaded. Second, it's not possible to unload a single assembly, so if unloading an assembly is required (to update a class, for example), it will need to be loaded in a separate application domain since application domains can be unloaded. For more information on application domains, consult the .NET CLR documentation.

Custom Code Generation

It's sometimes necessary for a class to have the best performance possible. For some algorithms, it's easy to write a general solution to a problem, but the overhead of the general solution may be undesirable. A custom solution to the problem may be possible but can't be generated ahead of time because the particulars of the problem aren't known until the runtime.

In such situations, it may be useful to generate the custom solution at execution time. This technique is often known as *self-modifying code*.

Polynomial Evaluation

This section implements a polynomial evaluator for polynomials in the following form:

```
Y = anxn + … + a2x2 + a1x + a0
```

To get rid of the exponentiation operation, which is slow, you can nicely rearrange the equation into this:

```
Y = a0 + x (a1 + x (a2 + … x (an) ) )
```

The first step in this exercise is to write the simple general solution to the problem. Since several solutions will exist, it will take a few files to build up a framework. The first is a utility class to do timing:

```
using System;
namespace Polynomial
{
    class Counter
    {
        public static long Frequency
        {
            get
            {
                long freq = 0;
                QueryPerformanceFrequency(ref freq);
                return freq;
            }
        }
        public static long Value
        {
            get
            {
                long count = 0;
                QueryPerformanceCounter(ref count);
                return count;
            }
        }

        [System.Runtime.InteropServices.DllImport("KERNEL32")]
        private static extern bool
            QueryPerformanceCounter(  ref long lpPerformanceCount);

        [System.Runtime.InteropServices.DllImport("KERNEL32")]
        private static extern bool
            QueryPerformanceFrequency( ref long lpFrequency);
    }
}
```

The `Counter` class encapsulates the Win32 performance counter functions and can be used to get accurate timings. .NET 2.0 introduces a new class called `Stopwatch` that wraps the functionality of the Win32 performance counter. If an application requires precise time functionality, and it doesn't require support for older framework versions, you should use `Stopwatch` over the `Counter` class.

Next, add a helper class to hold the information about `Polynomial`:

```
namespace Polynomial
{
    using System;
    using PolyInterface;

        /// <summary>
        /// The abstract class all implementations inherit from
        /// </summary>
    public abstract class Polynomial
    {
        public Polynomial(params double[] coefficients)
        {
            this.coefficients = new double[coefficients.Length];

            for (int i = 0; i < coefficients.Length; i++)
                this.coefficients[i] = coefficients[i];
        }

        public abstract double Evaluate(double value);
        public abstract IPolynomial GetEvaluate();

        protected double[] coefficients = null;
    }
}
```

The `Polynomial` class is an abstract class that holds the polynomial coefficients. A small interface defines the evaluation function:

```
namespace PolyInterface
{
        /// <summary>
        /// The interface that implementations will implement
        /// </summary>
    public interface IPolynomial
    {
        double Evaluate(double value);
    }
}
```

The following class implements the general method of evaluation:

```
namespace Polynomial
{
    using System;
    /// <summary>
    /// The simplest polynomial implementation
    /// </summary>
    /// <description>
    /// This implementation loops through the coefficients and evaluates each
```

```
    /// term of the polynomial.
    /// </description>
    class PolySimple: Polynomial
    {
        public PolySimple(params double[] coefficients): base(coefficients)
        {
        }

        public override IPolynomial GetEvaluate()
        {
            return((IPolynomial) this);
        }

        public override double Evaluate(double value)
        {
            double retval = coefficients[0];

            double f = value;

            for (int i = 1; i < coefficients.Length; i++)
            {
                retval += coefficients[i] * f;
                f *= value;
            }
            return(retval);
        }
    }
}
```

This is a simple evaluator that merely walks through the polynomial term by term, accumulates the values, and returns the result.

Finally, the driver ties it all together:

```
namespace Polynomial
{
    using System;
    using System.Diagnostics;

    /// <summary>
    /// Driver class for the project
    /// </summary>
    public class Driver
    {
        /// <summary>
        /// Times the evaluation of a polynomial
        /// </summary>
        /// <param name="p">The polynomial to evaluate</param>
```

```
public static double TimeEvaluate(Polynomial p)
{
    double value = 2.0;

    Console.WriteLine("{0}", p.GetType().Name);

        // Time the first iteration. This one is done
        // separately so that we can figure out the startup
        // overhead separately...
    long start = Stopwatch.GetTimeStamp();

    IPolynomial iPoly = p.GetEvaluate();
    long delta = Stopwatch.GetTimeStamp() -start;
    Console.WriteLine("Overhead = {0:f2} seconds",
          (double) delta/ Stopwatch.Frequency);
    Console.WriteLine("Eval({0}) = {1}", value, iPoly.Evaluate(value));

    int limit = 100000;
    Stopwatch timer = Stopwatch.StartNew();

        // Evaluate the polynomial the required number of
        // times.
    double result = 0;
    for (int i = 0; i < limit; i++)
    {
        result += iPoly.Evaluate(value);
    }
    timer.Stop();
    double ips = (double)limit / (double)timer.ElapsedMilliseconds / 1000D;
    Console.WriteLine("Evalutions/Second = {0:f0}", ips);
    Console.WriteLine();

    return(ips);
}
/// <summary>
/// Run all implementations for a given set of coefficients
/// </summary>
/// <param name="coeff"> </param>
public static void Eval(double[] coeff)
{
    Polynomial[] imps = new Polynomial []
    {
            new PolySimple(coeff),
    };
```

```
            double[] results = new double[imps.Length];
            for (int index = 0; index < imps.Length; index++)
            {
                results[index] = TimeEvaluate(imps[index]);
            }

            Console.WriteLine("Results for length = {0}", coeff.Length);
            for (int index = 0; index < imps.Length; index++)
            {
                Console.WriteLine("{0} = {1:f0}", imps[index], results[index]);
            }
            Console.WriteLine();
        }

        /// <summary>
        /// Main function.
        /// </summary>
        public static void Main()
        {
            Eval(new Double[] {5.5});

                // Evaluate the second polynomial, with 7 elements
            double[] coeff = new double[] {5.5, 7.0, 15, 30, 500, 100, 1};

            Eval(coeff);

                // Evaluate the second polynomial, with 50 elements
            coeff = new double[50];
            for (int index = 0; index < 50; index++)
            {
                coeff[index] = index;
            }
            Eval(coeff);
        }
    }
}
```

The `TimeEvaluate()` function takes a class that derives from `Polynomial` and calls `GetEvaluate()` to obtain the `IPolynomial` interface to do the evaluation. It times the `GetEvaluate()` function to determine the initialization overhead and then calls the evaluation function 100,000 times.

The driver evaluates polynomials with 1, 7, and 50 coefficients and writes out timing information.

The initial run generates the results in Table 32-1 (which are counts in evaluations/second).

Table 32-1. *Initial Results*

Method	c=1	c=7	c=50
Simple	18,000,000	6,400,000	1,600,000

Those results are really quite good, but it will be interesting to see if a custom solution can do better.

A Custom C# Class

The general solution has some loop overhead, and it'd be nice to get rid of this. To do this, you need a version of `eval` that evaluates an expression directly.

This example will generate a class that evaluates the polynomial in a single expression. It will generate a class like this:

```
// Polynomial evaluator
// Evaluating Y = 5.5 + 7 X^1 + 15 X^2 + 30 X^3 + 500 X^4 + 100 X^5 + 1 X^6
public class Poly_1001 : PolyInterface.IPolynomial {

    public double Eval(double x) {
        return (5.5
                    + (x * (7
                    + (x * (15
                    + (x * (30
                    + (x * (500
                    + (x * (100
                    + (x * (1 + 0))))))))))));
    }
}
```

The class to generate this file, compile it, and load it is as follows:

```
using System;
using System.IO;
using System.Diagnostics;
using System.Reflection;

class PolyCodeSlow: Polynomial
{
    public PolyCodeSlow(params double[] coefficients): base(coefficients)
    {
    }

    void WriteCode()
    {
        string timeString = polyNumber.ToString();
        polyNumber++;
```

```
string filename = "PS_" + timeString;
Stream s = File.Open(filename + ".cs", FileMode.Create);
StreamWriter t = new StreamWriter(s);

t.WriteLine("// polynomial evaluator");
t.Write("// Evaluating y = ");

string[] terms = new string[coefficients.Length];
terms[0] = coefficients[0].ToString();

for (int i = 1; i < coefficients.Length; i++)
    terms[i] = String.Format("{0} X^{1}", coefficients[1], i);

t.Write("{0}", String.Join(" + ", terms));
t.WriteLine();

t.WriteLine("");

string className = "Poly_" + timeString;
t.WriteLine("class {0}", className);
t.WriteLine("{");
t.WriteLine("public double Eval(double value)");
t.WriteLine("{");
t.WriteLine("    return(");
t.WriteLine("        {0}", coefficients[0]);

string closing = "";
for (int i = 1; i < coefficients.Length; i++)
{
    t.WriteLine("        + value * ({0} ", coefficients[i]);
    closing += ")";
}
t.Write("\t{0}", closing);
t.WriteLine(");");

t.WriteLine("}");
t.WriteLine("}");
t.Close();
s.Close();

//compile the DLL
CompilerParameters compParams = new CompilerParameters();
compParams.CompilerOptions = "/target:library /o+";
compParams.ReferencedAssemblies.AddRange(new string[]
 {Path.GetFileName(Assembly.GetExecutingAssembly().CodeBase),
  "mscorlib.dll",
  "System.dll"});
```

```
        compParams.IncludeDebugInformation = false;
        compParams.GenerateInMemory = false;
        compParams.OutputAssembly = (filename + ".dll");
        CompilerResults res = provider.CompileAssemblyFromFile(compParams,
            filename + ".cs");

            // Open the file, and get a pointer to the method info
         Assembly a = Assembly.LoadFrom(filename + ".dll");

         func = a.CreateInstance(className);

         invokeType = a.GetType(className);

         File.Delete(filename + ".cs");
    }
    public override IPolynomial GetEvaluate()
    {
        return((IPolynomial) this);
    }

    public override double Evaluate(double value)
    {
        object[] args = new Object[] {value};
        object retValue =
            invokeType.InvokeMember("Eval",
                    BindingFlags.Default | BindingFlags.InvokeMethod,
                    null,
                    func,
                    args);
        return((double) retValue);
    }

    object func = null;
    Type invokeType = null;

    static int polyNumber = 0;    // which number we're using...
}
```

The first time a polynomial is evaluated, the `WriteCode()` function writes the code out to the file and compiles it. It then uses `Assembly.LoadFrom()` to load the assembly and `Activator.CreateInstance()` to create an instance of the class. The instance and the type are then stored away for later use.

When it's time to call the function, the value for x is put into an array, and `Type.InvokeMember()` is used to locate and call the function.

When this version is called, it generates the results shown in Table 32-2.

Table 32-2. *Custom Class Results*

Method	c=1	c=7	c=50
Simple	18,000,000	6,400,000	1,600,000
Custom	43,000	43,000	41,000

These aren't exactly the results you want. The problem is that `Type.MethodInvoke()` is a general function and has a lot to do. It needs to locate the function based on the name and parameters and perform other operations, and it does this every time the function is called.

What's needed is a way to perform the call without the overhead—in other words, a way to define the way a method will look without defining what class the method is in. That's a perfect description of an interface.

A Fast Custom C# Class

Rather than calling the evaluate function directly like in the previous example, you can change the code so that the custom class implements the interface. After the assembly is loaded and an instance of the class is created, it will be cast to the interface, and that interface will be returned to the driver program.

This approach has two benefits. First, you'll call directly through the interface, so you avoid the overhead of `Type.InvokeMember()`. Second, instead of the driver calling the class evaluation function, which then called the custom function, the driver will call the custom function directly.

When you try this example, you get the results shown in Table 32-3.

Table 32-3. *Fast Custom Class Result*

Method	c=1	c=7	c=50
Simple	18,000,000	6,400,000	1,600,000
Custom	43,000	43,000	41,000
Custom fast	51,000,000	9,600,000	1,500,000

That gives a nice performance increase for small polynomials. For large ones, however, it turns out the function is so big that it doesn't fit into the cache and therefore gets slower than the simple method.

This has a problem, however. There's approximately half a second of overhead to write the file, compile it, and read in the resulting assembly. That's fine if each polynomial is evaluated many times but not if each one is evaluated only a few times.

Also, because this technique involves C# code, the compiler must be present on the system on which the code executes. Depending on where the application will run, this may be a problem.

What's needed is a way to get rid of the overhead and the dependency on the C# compiler.

A CodeDOM Implementation

A trip through the .NET Framework documentation shows a set of classes referred to as the *CodeDOM*. Visual Studio .NET designers use the CodeDOM to write the code for Windows Forms and Web Forms.

You can use a similar technique to generate a custom class for this example:

```
using System;
using System.IO;
using System.Diagnostics;
using System.Reflection;
using PolyInterface;
using System.CodeDom;
using System.CodeDom.Compiler;
using Microsoft.CSharp;

class PolyCodeDom: Polynomial
{
    public PolyCodeDom(params double[] coefficients): base(coefficients)
    {
    }

    void WriteCode()
    {
        string timeString = polyNumber.ToString();
        polyNumber++;

        string filename = "PSCD_" + timeString;
        Stream s = File.Open(filename + ".cs", FileMode.Create);
        StreamWriter t = new StreamWriter(s);

            // Generate code in C#
        CSharpCodeProvider provider = new CSharpCodeProvider();
        ICodeGenerator cg = provider.CreateGenerator(t);
        CodeGeneratorOptions op = new CodeGeneratorOptions();

            // Generate the comments at the beginning of the function
        CodeCommentStatement comment =
               new CodeCommentStatement("Polynomial evaluator");
        cg.GenerateCodeFromStatement(comment, t, op);

        string[] terms = new string[coefficients.Length];
        terms[0] = coefficients[0].ToString();

        for (int i = 1; i < coefficients.Length; i++)
            terms[i] = String.Format("{0} X^{1}", coefficients[i], i);
```

```
comment = new CodeCommentStatement(
        "Evaluating Y = " + String.Join(" + ", terms));
cg.GenerateCodeFromStatement(comment, t, op);

    // The class is named with a unique name
string className = "Poly_" + timeString;
CodeTypeDeclaration polyClass = new CodeTypeDeclaration(className);
    // The class implements IPolynomial
polyClass.BaseTypes.Add("PolyInterface.IPolynomial");

    // Set up the Eval function
CodeParameterDeclarationExpression param1 =
    new CodeParameterDeclarationExpression("double", "x");
CodeMemberMethod eval = new CodeMemberMethod();
eval.Name = "Evaluate";
eval.Parameters.Add(param1);

    // work-around for bug below...
eval.ReturnType = new CodeTypeReference("public double");
    // BUG: This doesn't generate "public", it just leaves
    // the attribute off of the member...
eval.Attributes |= MemberAttributes.Public;

    // Create the expression to do the evaluation of the
    // polynomial. To do this, we chain together binary
    // operators to get the desired expression
    // a0 + x * (a1 + x * (a2 + x * (a3)));
    //
    // This is very much like building a parse tree for
    // an expression.

CodeBinaryOperatorExpression plus = new CodeBinaryOperatorExpression();
plus.Left = new CodePrimitiveExpression(coefficients[0]);
plus.Operator = CodeBinaryOperatorType.Add;

CodeBinaryOperatorExpression current = plus;
for (int i = 1; i < coefficients.Length; i++)
{
    CodeBinaryOperatorExpression multiply =
        new CodeBinaryOperatorExpression();
    current.Right = multiply;
    multiply.Left = new CodeSnippetExpression("x");
    multiply.Operator = CodeBinaryOperatorType.Multiply;
```

```
            CodeBinaryOperatorExpression add = new CodeBinaryOperatorExpression();
            multiply.Right = add;
            add.Operator = CodeBinaryOperatorType.Add;
            add.Left = new CodePrimitiveExpression(coefficients[i]);
            current = add;
        }
        current.Right = new CodePrimitiveExpression(0.0);

            // return the expression...
        eval.Statements.Add(new CodeMethodReturnStatement(plus));
        polyClass.Members.Add(eval);
        cg.GenerateCodeFromType(polyClass, t, op);

        t.Close();
        s.Close();

       //compile the DLL
       CompilerParameters compParams = new CompilerParameters();
       compParams.CompilerOptions = "/target:library /o+";
       compParams.ReferencedAssemblies.AddRange(new string[]
        {Path.GetFileName(Assembly.GetExecutingAssembly().CodeBase),
         "mscorlib.dll",
         "System.dll"});
       compParams.IncludeDebugInformation = false;
       compParams.GenerateInMemory = false;
       compParams.OutputAssembly = (filename + ".dll");
       CompilerResults res = provider.CompileAssemblyFromFile(compParams,
           filename + ".cs");

            // Open the file, create the instance, and cast it
            // to the assembly
        Assembly a = Assembly.LoadFrom(filename + ".dll");
        polynomial = (IPolynomial) a.CreateInstance(className);

        File.Delete(filename + ".cs");
    }

    public override IPolynomial GetEvaluate()
    {
        if (polynomial == null)
            WriteCode();

        return((IPolynomial) polynomial);
    }
```

```
    public override double Evaluate(double value)
    {
        return(0.0);        // not used
    }

    IPolynomial polynomial = null;
    static int polyNumber = 1000;
}
```

Because the approach is the same, this technique yields similar performance.

A Reflection.Emit Implementation

With a bit more digging in the documentation, you may come across the `Reflection.Emit` namespace. Using the classes in this namespace, it's possible to create classes in memory and write the IL for the functions directly.

Using `Reflection.Emit` is fairly challenging because functions are written in the IL language rather than in C#. IL is roughly as difficult to develop in as assembly language, though the .NET IL is quite a bit simpler than x86 assembly language. The IL reference guide that ships with the SDK will be a useful reference.[1]

The easiest way to determine what IL to generate is to write the class in C#, compile it, and then use ILDASM to figure out what IL to generate. To evaluate the polynomial expression, the C# compiler uses a regular pattern, so generating the IL is straightforward.

Here's the class that does it:

```
using System;
using System.IO;
using System.Diagnostics;
using System.Reflection;
using System.Reflection.Emit;
using System.Threading;
using PolyInterface;

class PolyEmit: Polynomial
{
    Type    theType = null;
    object    theObject = null;
    IPolynomial poly = null;

    public PolyEmit(params double[] coefficients): base(coefficients)
    {
    }
        /// <summary>
        /// Create an assembly that will evaluate the polynomial.
        /// </summary>
```

1. Refer to `C:\Program Files\Microsoft.Net\FrameworkSDK\Tool Developers Guide`.

```
private Assembly EmitAssembly()
{
        //
        // Create an assembly name
        //
    AssemblyName assemblyName = new AssemblyName();
    assemblyName.Name = "PolynomialAssembly";

        //
        // Create a new assembly with one module
        //
    AssemblyBuilder newAssembly =
        Thread.GetDomain().DefineDynamicAssembly(
                assemblyName, AssemblyBuilderAccess.Run);
    ModuleBuilder newModule = newAssembly.DefineDynamicModule("Evaluate");

        //
        //  Define a public class named "PolyEvaluate" in the assembly.
        //
    TypeBuilder myType =
        newModule.DefineType("PolyEvaluate", TypeAttributes.Public);

        //
        // Mark the class as implementing IPolynomial. This is
        // the first step in that process.
        //
    myType.AddInterfaceImplementation(typeof(IPolynomial));

        // Add a constructor
    ConstructorBuilder constructor =
        myType.DefineDefaultConstructor(MethodAttributes.Public);

        //
        // Define a method on the type to call. We pass an
        // array that defines the types of the parameters,
        // the type of the return type, the name of the method,
        // and the method attributes.
        //
    Type[] paramTypes = new Type[] {typeof(double)};
    Type returnType = typeof(double);
    MethodBuilder simpleMethod =
        myType.DefineMethod("Evaluate",
                        MethodAttributes.Public | MethodAttributes.Virtual,
                        returnType,
                        paramTypes);
```

```
    //
    // From the method, get an ILGenerator. This is used to
    // emit the IL that we want.
    //
ILGenerator il = simpleMethod.GetILGenerator();

    //
    // Emit the IL. This is a hand-coded version of what
    // you'd get if you compiled the code example and then ran
    // ILDASM on the output.
    //

    //
    // This first section repeated loads the coefficient's
    // x value on the stack for evaluation.
    //
for (int index = 0; index < coefficients.Length - 1;index++)
{
    il.Emit(OpCodes.Ldc_R8, coefficients[index]);
    il.Emit(OpCodes.Ldarg_1);
}

    // load the last coefficient
il.Emit(OpCodes.Ldc_R8, coefficients[coefficients.Length - 1]);

    // Emit the remainder of the code. This is a repeated
    // section of multiplying the terms together and
    // accumulating them.
for (int loop = 0; loop < coefficients.Length - 1; loop++)
{
    il.Emit(OpCodes.Mul);
    il.Emit(OpCodes.Add);
}

    // return the value
il.Emit(OpCodes.Ret);

    //
    // Finish the process.
    // Create the type.
    //
//myType.CreateType();
```

```
            //
            // Hook up the interface member to the member function
            // that implements that member.
            // 1) Get the interface member.
            // 2) Hook the method to the interface member.
            //
        MethodInfo methodInterfaceEval = typeof(IPolynomial).GetMethod("Evaluate");

        myType.DefineMethodOverride(simpleMethod, methodInterfaceEval);
        myType.CreateType();

          return newAssembly;
    }

    public void Setup()
    {
            // Create the assembly, create an instance of the
            // evalution class, and save away an interface
            // reference to it.
        Assembly ass = EmitAssembly();

        theObject = ass.CreateInstance("PolyEvaluate");
        theType = theObject.GetType();

        poly = (IPolynomial) theObject;
    }

    public override IPolynomial GetEvaluate()
    {
        if (theType == null)
            Setup();

        return((IPolynomial) poly);
    }

    public override double Evaluate(double value)
    {
        return(0.0f);
    }
}
```

The best way to understand this code is to look at the ILDASM for the previous example, walk through the code, look up the classes in the documentation, and read the comments.

The implementation using `Reflection.Emit` has nearly identical performance to the other fast techniques but less overhead (about 0.25 seconds for the first polynomial and no measurable overhead for later ones). Table 32-4 shows the final results.

Table 32-4. *Results Summary*

Method	c=1	c=7	c=50
Simple	18,000,000	6,400,000	1,600,000
Custom	43,000	43,000	41,000
Custom fast	51,000,000	9,600,000	1,500,000
CodeDOM	51,000,000	9,600,000	1,500,000
`Reflection.Emit`	51,000,000	9,600,000	1,500,000

Lightweight Code Generation

An assembly can't be unloaded from an application domain once it has been loaded. This means the only way to unload an assembly is to unload the entire application domain, which generally means if an assembly needs to be unloaded without terminating a process, a secondary application domain needs to be created just to hold the assemblies that need to be unloaded. In the examples presented in this chapter so far, the dynamically created assemblies are loaded into the main application domain and hence can't be unloaded.

■ **Note** For a code example of using a secondary application domain for assembly unloading, see *Maximizing .NET Performance* (Apress, 2003) by Nick Wienholt.

In some situations, dynamically generated code is quite small in nature, and the requirement of generating a full assembly and then housing this in a secondary application domain just so it can be unloaded feels like overkill. In recognition of this, .NET 2.0 introduces a lightweight code generation model that allows the generation of static methods without a full assembly, without the requirement to undergo verification (security permissions permissible), and with the ability to be reclaimed without unloading an application domain.

Because the methods generated by lightweight code generation must be static, they couldn't be used to implement the polynomial examples presented earlier in this chapter that relied on deriving from the abstract `Polynomial` class; however, by removing the polymorphic and instance calls, it's possible to solve polynomials with lightweight code generation:

```
using System;
using System.Reflection;
using System.Reflection.Emit;

namespace Polynomial
{
 class LightWeightPoly
 {
  public void Eval()
  {
   // Evaluate the first polynomial, with 7 elements
   double[] coeff = new double[] { 5.5, 7.0, 15, 30, 500, 100, 1 };
   DynamicMethod dm = GetEvaluator(coeff);

   object[] parameter = new object[] { 2.0 };
   double result = (double)dm.Invoke(null, parameter);
  }

  DynamicMethod GetEvaluator(params double[] coefficients)
  {
   //define dynamic method contruction data
   Type[] paramTypes = new Type[] { typeof(double) };
   Type returnType = typeof(double);
   Type methodOwner = this.GetType();
   //
   //create dynamic method
   DynamicMethod dm = new DynamicMethod("Evaluate", returnType,
    paramTypes, methodOwner, false);
   ILGenerator il = dm.GetILGenerator();
   //
   // Emit the IL. This is a hand-coded version of what
   // you'd get if you compiled the code example and then ran
   // ILDASM on the output.
   //
   //
   // This first section repeated loads the coefficient's
   // x value on the stack for evaluation.
   //
   for (int index = 0; index < coefficients.Length - 1; index++)
   {
    il.Emit(OpCodes.Ldc_R8, coefficients[index]);
    il.Emit(OpCodes.Ldarg_1);
   }
   // load the last coefficient
   il.Emit(OpCodes.Ldc_R8, coefficients[coefficients.Length - 1]);
   // Emit the remainder of the code. This is a repeated
   // section of multiplying the terms together and
```

```
    // accumulating them.
    for (int loop = 0; loop < coefficients.Length - 1; loop++)
    {
     il.Emit(OpCodes.Mul);
     il.Emit(OpCodes.Add);
    }
    // return the value
    il.Emit(OpCodes.Ret);

    //dynamic method now done - return it
    return dm;
   }
  }
}
```

Generating C# code and compiling it or using `Reflection.Emit` are valid techniques, but they should probably be the last resort in your performance improvement arsenal. The simple example in this chapter is much easier to write, debug, and maintain.

Techniques such as these are already used in the .NET Framework; the regular expression class in the `System.Text.RegularExpressions` namespace uses `Reflection.Emit` to generate a custom-matching engine when a regular expression is compiled.

CHAPTER 33

Interop

One of the important capabilities of C# is being able to interoperate with existing code, whether it's based on COM or is in a native DLL. This chapter provides a brief overview of how interop works.

Tip For more information on interop, refer to *COM and .NET Interoperability* (Apress, 2002) by Andrew Troelsen or *.NET and COM: The Complete Interoperability Guide* (Sams, 2002) by Adam Nathan.

Using COM Objects

To call a COM object, the first step is to define a proxy (or *wrapper*) class that specifies the functions in the COM object, along with additional information. This is a fair amount of work, which you can avoid in most cases by using the `tlbimp` utility. This utility reads the COM typelib information and then creates the proxy class automatically. This will work in many situations, but if you need more control over marshalling, you may have to write the proxy class by hand. In this case, you use attributes to specify how marshalling should be performed.

Once the proxy class is written, it's used like any other .NET class, and the runtime handles the ugly stuff.

Being Used by COM Objects

The runtime also lets .NET objects be used in place of COM objects. The `tlbexp` utility creates a typelib that describes the COM objects so that other COM-based programs can determine the object's interface, and the `regasm` utility registers an assembly so it can be accessed through COM. When COM accesses a .NET class, the runtime creates the .NET object, fabricating whatever COM interfaces are required and marshalling the data between the .NET world and the COM world.

Calling Native DLL Functions

C# can call C functions written in native code through a runtime feature known as *platform invoke*. The `DllImport` attribute specifies the file that the function is located in and can also

specify the default character marshalling. In many cases, that attribute is all you need, but if a value is passed by reference, you can specify `ref` or `out` to tell the marshaller how to pass the value. Here's an example:

```
using System;
using System.Runtime.InteropServices;
class Test
{
    [DllImport("user32.dll")]
    public static extern int MessageBox(IntPtr h, string m,
        string c, int type);
    public static void Main()
    {
        int retval = MessageBox(IntPtr.Zero, "Hello", "Caption", 0);
    }
}
```

When this code runs, a message box will appear. Note that the code uses `MessageBox()` rather than the ASCII- or Unicode-specific versions; the runtime will automatically use the appropriate function (`MessageBoxA()` or `MessageBoxW()`) based on the platform, but you can specify up front exactly which variant to pick.

`IntPtr`, which is used as the type for the first parameter in the `MessageBox()` call, represents pointers or handles that are platform-specific. The advantage of `IntPtr` over raw integral types is that `IntPtr` is defined to match the pointer size of the underlying platform, making conversion to 64-bit (or wider) platforms much easier.

C# can't use C++ classes directly; to use such objects, they must be exposed in a .NET-compliant way using the C++/CLI or as COM objects.

Pointers and Declarative Pinning

It's common for C-style functions to take pointers as their parameters. Since C# supports pointers in an unsafe context, it's straightforward to use such functions. This example calls `ReadFile()` from `kernel32.dll`:

```
// file=ReadFileUnsafe.cs
// compile with: csc /unsafe ReadFileUnsafe.cs
using System;
using System.Runtime.InteropServices;
using System.Text;

class FileRead
{
    const uint GENERIC_READ = 0x80000000;
    const uint OPEN_EXISTING = 3;
    IntPtr handle;
```

```
    public FileRead(string filename)
    {
        // opens the existing file...
        handle = CreateFile(    filename,
                GENERIC_READ,
                0,
                0,
                OPEN_EXISTING,
                0,
                0);
    }

    [DllImport("kernel32", SetLastError=true)]
    static extern IntPtr CreateFile(
        string filename,
        uint desiredAccess,
        uint shareMode,
        uint attributes,        // really SecurityAttributes pointer
        uint creationDisposition,
        uint flagsAndAttributes,
        uint templateFile);

// SetLastError =true is used to tell the interop layer to keep track of
//underlying Windows errors
    [DllImport("kernel32", SetLastError=true)]
    static extern unsafe bool ReadFile(
        IntPtr hFile,
        void* lpBuffer,
        int nBytesToRead,
        int* nBytesRead,
        int overlapped);

    public unsafe int Read(byte[] buffer, int count)
    {
        int n = 0;
        fixed (byte* p = buffer)
        {
            ReadFile(handle, p, count, &n, 0);
        }
        return n;
    }
}
```

```
class Test
{
    public static void Main(string[] args)
    {
        FileRead fr = new FileRead(args[0]);

        byte[] buffer = new byte[128];
        ASCIIEncoding e = new ASCIIEncoding();

            // loop through, read until done...
        Console.WriteLine("Contents");
        while (fr.Read(buffer, 128) != 0)
        {
            Console.Write("{0}", e.GetString(buffer));
        }
    }
}
```

In this example, the `FileRead` class encapsulates the code to read from the file. It declares the functions to import and the `unsafe Read` function.

Calling the `ReadFile()` function presents a dilemma. The `byte[]` array buffer is a managed variable, which means the garbage collector can move it at any time. But `ReadFile()` expects that the buffer pointer passed to it won't move during the call to `ReadFile()`.

The `fixed` statement bridges the two worlds. It "pins" the `byte[]` buffer by setting a flag on the object so the garbage collector won't move the object if a collection occurs inside the `fixed` block. This makes it safe to pass the pointer to the buffer to `ReadFile()`. After the call, the flag is cleared and execution continues.

This approach is nice in that it has low overhead—unless a garbage collection occurs while the code is inside the `fixed` block, which is unlikely.

This sample works fine, but the class is subject to the usual constraints on code written with `unsafe`. See Chapter 38 for more information.

A Safe Version

Since pointer support isn't required for .NET languages, other languages need to be able to call functions such as `ReadFile()` without using pointers. The runtime provides a considerable amount of support to make the marshalling from managed to unmanaged types (including pointer types) transparent.

Therefore, you can rewrite the previous example without using `unsafe`. All that's required is to change the `extern` declaration for `ReadFile()` and the `Read()` function:

```
[DllImport("kernel32", SetLastError=true)]
static extern bool ReadFile(
    IntPtr hFile,
    byte[] buffer,
    int nBytesToRead,
    ref int nBytesRead,
    int overlapped);
```

```
public int Read(byte[] buffer, int count)
{
    int n = 0;
    ReadFile(handle, buffer, count, ref n, 0);
    return n;
}
```

In this code, the pointer parameter for the buffer has been changed to a `byte[]`, and the number of characters read is defined as a `ref int` instead of an `int*`.

In this version, the runtime will do the pinning of the buffer automatically rather than having to be explicit, and because `unsafe` isn't required, this version isn't subject to the same restrictions as the previous example.

Structure Layout

The runtime allows a structure to specify the layout of its data members by using the `StructLayout` attribute. By default, the layout of a structure is automatic, which means the runtime is free to rearrange the fields. When using interop to call into native or COM code, you require better control.

When specifying the `StructLayout` attribute, you can specify three kinds of layout using the `LayoutKind` enum:

- `Auto`, where the runtime chooses the appropriate way to lay out the members.
- `Sequential`, where all fields are in declaration order. For sequential layout, you can use the `Pack` property to specify the type of packing.
- `Explicit`, where every field has a specified offset. In explicit layout, the `StructOffset` attribute must be used on every member to specify the offset in bytes of the element.

Additionally, you can specify the `CharSet` property to set the default marshalling for string data members.

By default, the C# compiler sets sequential layout for all structs.

Calling a Function with a Structure Parameter

To call a function with a structure parameter, you need to define the structure with the appropriate parameters. This example shows how to call `GetWindowPlacement()`:

```
using System;
using System.Runtime.InteropServices;

struct Point
{
    public int x;
    public int y;

    public override string ToString()
    {
        return(String.Format("({0}, {1})", x, y));
    }
}
```

```
struct Rect
{
    public int left;
    public int top;
    public int right;
    public int bottom;

    public override string ToString()
    {
        return(String.Format("({0}, {1})\n    ({2}, {3})",
                    left, top, right, bottom));
    }
}

struct WindowPlacement
{
    public uint length;
    public uint flags;
    public uint showCmd;
    public Point minPosition;
    public Point maxPosition;
    public Rect normalPosition;

    public override string ToString()
    {
        return(String.Format("min, max, normal:\n{0}\n{1}\n{2}",
            minPosition, maxPosition, normalPosition));
    }
}

class Window
{
    [DllImport("user32")]
    static extern IntPtr GetForegroundWindow();

    [DllImport("user32")]
    static extern bool GetWindowPlacement(IntPtr handle, ref WindowPlacement wp);

    public static void Main()
    {
        IntPtr window = GetForegroundWindow();

        WindowPlacement wp = new WindowPlacement();
        wp.length = (uint) Marshal.SizeOf(wp);
```

```
        bool result = GetWindowPlacement(window, ref wp);

        if (result)
        {
            Console.WriteLine(wp);
        }
    }
}
```

Fixed-Size Buffers

It's common practice for a C language application to use fixed-sized buffers to store data, both in memory and on disk. A fixed-sized buffer of some primitive type such as `int` or `char` is easy and quick to populate—the data that's needed to populate the entire buffer can be copied over the entire buffer using the C runtime `memcpy` or an equivalent command, populating all the elements in the buffer in a single operation. The simplicity and speed of accessing fixed-sized buffers comes at a considerable cost in terms of code correctness and security and is notorious as the source of many serious security breaches, which is why .NET uses a different model.

In the .NET model, all access to elements in a buffer is checked, and because arrays are reference types, an array declared as part of a structure doesn't physically live inside the structure. Instead, a reference to the array is placed inside the structure, which points to the location of the array on the heap. This means that a `memcpy` or equivalent wouldn't work (even if legal), as the memory inside the buffer isn't laid out in memory correctly. This causes headaches and inefficiencies when dealing in heavy interop scenarios.

In C# 2.0, it's now possible to overcome this problem with an extension to the `fixed` keyword, which allows arrays to be declared as fixed-sized inside unsafe code blocks. Fixed arrays will typically be part of a structure that's passed to a native API. The Windows API function `GetVersionEx` is a good example of an API where you can use fixed-sized buffers. The single parameter that's passed to the function is a pointer to an `OSVERSIONINFO` structure defined in C as follows:

```
typedef struct _OSVERSIONINFO {
 DWORD dwOSVersionInfoSize;
 DWORD dwMajorVersion;
 DWORD dwMinorVersion;
 DWORD dwBuildNumber;
 DWORD dwPlatformId;
 TCHAR szCSDVersion[128];
} OSVERSIONINFO;
```

Although you could call this function without fixed-sized buffers by using marshalling attributes, you'll receive a performance penalty. To convert this structure to a C# struct, use the following declaration:

```
unsafe struct OSVERSIONINFO
{
 public uint dwOSVersionInfoSize;
 public uint dwMajorVersion;
 public uint dwMinorVersion;
 public uint dwBuildNumber;
 public uint dwPlatformId;
 public fixed char szCSDVersion[128];
}
```

You can now call the `GetVersionEx` function using the `OSVERSIONINFO` buffer without any marshalling between the C# code and Windows API:

```
[DllImport("Kernel32.dll", CharSet = CharSet.Unicode)]
 static extern bool GetVersionEx(ref OSVERSIONINFO lpVersionInfo);

unsafe static void Main(string[] args)
{
 OSVERSIONINFO versionInfo = new OSVERSIONINFO();
 versionInfo.dwOSVersionInfoSize = (uint)sizeof(OSVERSIONINFO);
 bool res = GetVersionEx(ref versionInfo);
 Console.WriteLine(Marshal.PtrToStringUni(new IntPtr(versionInfo.szCSDVersion)));
}
```

As with all unsafe code, this involves a risk, and if the size of memory blocks don't line up correctly, you have the potential for application crashes and security vulnerabilities.

Hooking Up to a Windows Callback

The Win32 API sometimes uses callback functions to pass information to the caller asynchronously. The closest analogy to a callback function in C# (and in the CLR) is a delegate, so the runtime interop layer can map a delegate to a callback. Here's an example that does this for the `SetConsoleHandler()` API (the one used to catch Ctrl+C):

```
using System;
using System.Threading;
using System.Runtime.InteropServices;

class ConsoleCtrl
{
    public enum ConsoleEvent
    {
        CTRL_C = 0,        // From wincom.h
        CTRL_BREAK = 1,
        CTRL_CLOSE = 2,
        CTRL_LOGOFF = 5,
        CTRL_SHUTDOWN = 6
    }
```

```
    public delegate void ControlEventHandler(ConsoleEvent consoleEvent);

    public event ControlEventHandler ControlEvent;

        // save delegate so the GC doesn't collect it.
    ControlEventHandler eventHandler;

    public ConsoleCtrl()
    {
            // save this to a private var so the GC doesn't collect it...
        eventHandler = new ControlEventHandler(Handler);
        SetConsoleCtrlHandler(eventHandler, true);
    }

    private void Handler(ConsoleEvent consoleEvent)
    {
        if (ControlEvent != null)
            ControlEvent(consoleEvent);
    }

    [DllImport("kernel32.dll")]
    static extern bool SetConsoleCtrlHandler(ControlEventHandler e, bool add);
}

class Test
{
    public static void MyHandler(ConsoleCtrl.ConsoleEvent consoleEvent)
    {
        Console.WriteLine("Event: {0}", consoleEvent);
    }

    public static void Main()
    {
        ConsoleCtrl cc = new ConsoleCtrl();
        cc.ControlEvent += new ConsoleCtrl.ControlEventHandler(MyHandler);

        Console.WriteLine("Enter 'E' to exit");

        Thread.Sleep(15000);  // sleep 15 seconds
    }
}
```

The `ConsoleCtrl` class encapsulates the API function. It defines a delegate that matches the signature of the Win32 callback function and then uses that as the type passed to the Win32 function. It exposes an event that other classes can hook up to.

The one subtlety of this example has to do with the following line:

```
ControlEventHandler eventHandler;
```

This line is required because the interop layer will pass a pointer to the delegate to the Win32 function, but the garbage collector has no way to know that pointer exists. If the delegate isn't stored in a place the garbage collector can find, it's collected the next time the garbage collector runs, which is bad.

Design Guidelines

The following sections highlight a few guidelines to help you decide what method of interop to use and how to use it.

C# or C++?

The two options for doing interop with existing C libraries are calling functions directly from C# using platform invoke and using C++/CLI to encapsulate the C functions in a nice managed class written in C++.

Which is the better choice depends upon the interface being called. It's easy to tell by how much effort it takes to get it working. If it's straightforward, then doing it in C# is easy. If you find yourself wondering how to do that in C#, or you start using a lot of unsafe code, then it's likely that doing it using the Managed Extensions is a better choice. This is especially true for complex interfaces, where a structure contains pointers to other structures or the sizes of a structure aren't fixed.

In such cases, you'll need to do a translation from the C-style way of doing things to the .NET-managed way of doing things. This might involve grabbing a value out of a union or walking through a structure of variable size and then putting the values into the appropriate variable on a collection class. Doing such an operation is a lot easier in C++ than it is in C#, and you likely already have working C++ code on which you can base your C# code.

Marshalling in C#

When defining functions in C#, consider the following guidelines:

- In general, the data marshalling layer does the right thing. Choose the type that's closest to the type you want.
- For opaque types (such as pointers) where all you really care about is the size of the variable, just use an `IntPtr`.
- To control data marshalling, use the `MarshalAs` attribute. This is most often used to control string marshalling.
- Rather than using a pointer type for a parameter, define it using `ref` or `out`.
- Read about data marshalling in the .NET Framework's developer specifications.
- If things get ugly, switch to using C++/CLI. Switching to C++/CLI can range from writing a small wrapper that's then called from C# to using C++/CLI for large portions of an application the optimum technique depends on the skills of the developers available.

CHAPTER 34

.NET Framework Overview

The .NET Framework contains many functions normally found in language-specific runtime libraries, and it's therefore important to understand what classes are available in the .NET Framework.

Numeric Formatting

You format numeric types through the `Format()` member function of that data type. You can call this directly through `String.Format()`, which calls the `Format()` function of each data type, or through `Console.WriteLine()`, which calls `String.Format()`.

The "Custom Object Formatting" section later in this chapter covers how to add formatting to a user-defined object. The following sections discuss how you do formatting with the built-in types.

You have two methods of specifying numeric formatting. You can use a standard format string to convert a numeric type to a specific string representation. If you need further control over the output, you can use a custom format string.

Standard Format Strings

A standard format string consists of a character that specifies the format, followed by a sequence of digits that specifies the precision. The formats listed in Table 34-1 are supported.

Table 34-1. *Standard Format Strings*

Format Character	Description
`C`, `c`	Currency
`D`, `d`	Decimal
`E`, `e`	Scientific (exponential)
`F`, `f`	Fixed-point
`G`, `g`	General
`N`, `n`	Number
`R`, `r`	Round-trip
`X`, `x`	Hexadecimal

Currency

The currency format string converts the numerical value to a string containing a locale-specific currency amount. By default, the current locale determines the format information, but you can change this by passing a `NumberFormatInfo` object. This example:

```
using System;
class Test
{
    public static void Main()
    {
        Console.WriteLine("{0:C}", 33345.8977);
        Console.WriteLine("{0:C}", -33345.8977);
    }
}
```

gives the following output:

```
$33,345.90
($33,345.90)
```

An integer following the `c` specifies the number of decimal places to use; two places are used if the integer is omitted.

Decimal

The decimal format string converts the numerical value to an integer. The precision specifier determines the minimum number of digits. The result is left-padded with zeroes to obtain the required number of digits. This example:

```
using System;
class Test
{
    public static void Main()
    {
        Console.WriteLine("{0:D}", 33345);
        Console.WriteLine("{0:D7}", 33345);
    }
}
```

gives the following output:

```
33345
0033345
```

Scientific (Exponential)

The scientific (exponential) format string converts the value to a string in the following form:

```
m.dddE+xxx
```

One digit always precedes the decimal point, and you specify the number of decimal places with the precision specifier, with six places used as the default. The format specifier controls whether `E` or `e` appears in the output. This example:

```
using System;
class Test
{
    public static void Main()
    {
        Console.WriteLine("{0:E}", 33345.8977);
        Console.WriteLine("{0:E10}", 33345.8977);
        Console.WriteLine("{0:e4}", 33345.8977);
    }
}
```

gives the following output:

```
3.334590E+004
3.3345897700E+004
3.3346e+004
```

Fixed-Point

The fixed-point format string converts the value to a string, with the number of places after the decimal point specified by the precision specifier. Use two places if the precision specifier is omitted. This example:

```
using System;
class Test
{
    public static void Main()
    {
        Console.WriteLine("{0:F}", 33345.8977);
        Console.WriteLine("{0:F0}", 33345.8977);
        Console.WriteLine("{0:F5}", 33345.8977);
    }
}
```

gives the following output:

```
33345.90
33346
33345.89770
```

General

The general format string converts the value to either a fixed-point format or a scientific format, whichever one gives a more compact format. This example:

```
using System;
class Test
{
    public static void Main()
    {
        Console.WriteLine("{0:G}", 33345.8977);
        Console.WriteLine("{0:G7}", 33345.8977);
        Console.WriteLine("{0:G4}", 33345.8977);
    }
}
```

gives the following output:

```
33345.8977
33345.9
3.335E4
```

Number

The number format string converts the value to a number that has embedded commas:

```
12,345.11
```

By default, the number is formatted with two digits to the right of the decimal point. You can control this by specifying the number of digits after the format specifier. This example:

```
using System;
class Test
{
    public static void Main()
    {
        Console.WriteLine("{0:N}", 33345.8977);
        Console.WriteLine("{0:N4}", 33345.8977);
    }
}
```

gives the following output:

```
33,345.90
33,345.8977
```

It's possible to control the character used for the decimal point by passing a `NumberFormatInfo` object to the `Format()` function.

Round-Trip

Using the round-trip format ensures that a value represented as a string can be parsed back into the same value. This is especially useful for floating-point types, where extra digits are sometimes required to read a number back in with the identical value.

Hexadecimal

The hexadecimal format string converts the value to hexadecimal format. You set the minimum number of digits with the precision specifier; the number will be zero-padded to that width.

Using `X` will result in uppercase letters in the converted value; `x` will result in lowercase letters. This example:

```
using System;
class Test
{
    public static void Main()
    {
        Console.WriteLine("{0:X}", 255);
        Console.WriteLine("{0:x8}", 1456);
    }
}
```

gives the following output:

```
FF
000005b0
```

NumberFormatInfo

The `NumberFormatInfo` class controls the formatting of numbers. By setting the properties in this class, you can control the currency symbol, decimal separator, and other formatting properties.

Custom Format Strings

You can use custom format strings to obtain more control over the conversion than is available through the standard format strings. In custom format strings, special characters form a template into which the number is formatted. Any characters that don't have a special meaning in the format string are copied verbatim to the output. Table 34-2 describes the custom strings available.

Table 34-2. *String Format Characters*

Character	Description	Result
0	Displays zero placeholder	Displays leading zero if a number has fewer digits than there are zeroes in the format.
#	Displays digit placeholder	Replaces # with the digit only for significant digits.
.	Decimal point	Displays the decimal point.
,	Group separator and multiplier	Separates number groups, such as 1,000. When this is used after a number, it divides the number by 1,000.
%	Displays % notation	Displays the percent character.
;	Section separator	Uses different formats for positive, negative, and zero values.

Digit or Zero Placeholder

You can use the zero (0) character as a digit or as a zero placeholder. If the numeric value has a digit in the position at which the 0 appears in the format string, the digit will appear in the result. If not, a zero appears in that position. This example:

```
using System;
class Test
{
    public static void Main()
    {
        Console.WriteLine("{0:000}", 55);
        Console.WriteLine("{0:000}", 1456);
    }
}
```

gives the following output:

```
055
1456
```

Digit or Space Placeholder

You can use the pound (#) character as the digit or space placeholder. It works the same as the 0 placeholder, except that the character is omitted if there's no digit in that position. This example:

```
using System;
class Test
{
    public static void Main()
    {
        Console.WriteLine("{0:#####}", 255);
        Console.WriteLine("{0:#####}", 1456);
        Console.WriteLine("{0:###}", 32767);
    }
}
```

gives the following output:

```
255
1456
32767
```

Decimal Point

The first period (`.`) character that appears in the format string determines the location of the decimal separator in the result. A `NumberFormatInfo` instance controls the character used as the decimal separator in the formatted string. This example:

```
using System;
class Test
{
    public static void Main()
    {
        Console.WriteLine("{0:#####.000}", 75928.3);
        Console.WriteLine("{0:##.000}", 1456.456456);
    }
}
```

gives the following output:

```
75928.300
1456.456
```

Group Separator

You can use the comma (`,`) character as a group separator. If a comma appears in the middle of a display digit placeholder and to the left of the decimal point (if present), a group separator will be inserted in the string. A `NumberFormatInfo` instance controls the character used in the formatted string and the number of numbers to group. This example:

```
using System;
class Test
{
    public static void Main()
    {
        Console.WriteLine("{0:##,###}", 2555634323);
        Console.WriteLine("{0:##,000.000}", 14563553.593993);
        Console.WriteLine("{0:#,#.000}", 14563553.593993);
    }
}
```

gives the following output:

```
2,555,634,323
14,563,553.594
14,563,553.594
```

Number Prescaler

You can also use the comma (,) character to indicate that the number should be prescaled. In this usage, the comma must come directly before the decimal point or at the end of the format string.

For each comma present in this location, the number is divided by 1,000 before it's formatted. This example:

```
using System;
class Test
{
    public static void Main()
    {
        Console.WriteLine("{0:000,.##}", 158847);
        Console.WriteLine("{0:000,,,.###}", 1593833);
    }
}
```

gives the following output:

```
158.85
000.002
```

Percent Notation

You can use the percent (%) character to indicate that the number to be displayed should be displayed as a percentage. The number is multiplied by 100 before it's formatted. This example:

```
using System;
class Test
{
    public static void Main()
    {
        Console.WriteLine("{0:##.000%}", 0.89144);
        Console.WriteLine("{0:00%}", 0.01285);
    }
}
```

gives the following output:

```
89.144%
01%
```

Exponential Notation

When `E+0`, `E-0`, `e+0`, or `e-0` appear in the format string directly after a `#` or `0` placeholder, the number will be formatted in exponential notation. The number of digits in the exponent is controlled by the number of `0` placeholders that appear in the exponent specifier. The `E` or `e` is copied directly into the formatted string, and a plus (`+`) sign means there will be a plus or minus sign in that position, while a minus (`-`) sign means there's a character there only if the number is negative. This example:

```
using System;
class Test
{
    public static void Main()
    {
        Console.WriteLine("{0:###.000E-00}", 3.1415533E+04);
        Console.WriteLine("{0:#.0000000E+000}", 2.553939939E+101);
    }
}
```

gives the following output:

```
314.155E-02
2.5539399E+101
```

Section Separator

You can use the semicolon (`;`) character to specify different format strings for a number, depending on whether the number is positive, negative, or zero. If there are only two sections, the first section applies to positive and zero values and the second applies to negative values. If there are three sections, they apply to positive values, negative values, and the zero value. This example:

```
using System;
class Test
{
    public static void Main()
    {
        Console.WriteLine("{0:###.00;0;(###.00)}", -456.55);
        Console.WriteLine("{0:###.00;0;(###.00)}", 0);
        Console.WriteLine("{0:###.00;0;(###.00)}", 456.55);
    }
}
```

gives the following output:

```
457
(.00)
456.55
```

Escapes and Literals

You can use the backslash (\) character to escape characters so they aren't interpreted as formatting characters. Because the backslash already has meaning within C# literals, it will be easier to specify the string using the verbatim literal syntax; otherwise, a double backslash (\\) is required to generate a single slash in the output string.

You can specify a string of uninterpreted characters by enclosing them in single quotes; this may be more convenient than using the backslash character. This example:

```
using System;
class Test
{
    public static void Main()
    {
        Console.WriteLine("{0:###\\#}", 255);
        Console.WriteLine(@"{0:###\#}", 255);
        Console.WriteLine("{0:###'#0%;'}", 1456);
    }
}
```

gives the following output:

```
255#
255#
1456#0%;
```

Numeric Parsing

If there's a reasonable chance that the input string contains invalid characters, which means that `Parse` will be unable to convert to the appropriate type and throw an exception, you should use the `TryParse` method instead. Rather than throwing an exception if the input can't be successfully converted, `TryParse` instead returns a Boolean that indicates the success of the conversion, with the result of the conversion returned as an `out` parameter:

```
Console.WriteLine("Please enter an integer and press Enter");
int numberEntered;
while(!int.TryParse(Console.ReadLine(), out numberEntered)){
 Console.WriteLine("Please try again");
}
Console.WriteLine("You entered " + numberEntered.ToString());
```

In the .NET Framework versions 1.0 and 1.1, `double` was the only type that had a `TryParse` method. For 2.0, the number of types that offer a `TryParse` method has been expanded and now includes all numeric types, `char`, `DateTime`, and `TimeSpan`.

Date and Time Formatting

The `DateTime` class provides flexible formatting options. You can specify several single-character formats; the class also supports custom formatting. Table 34-3 indicates the standard `DateTime` formats.

Table 34-3. *`DateTime` Format Strings*

Character	Pattern	Description
`d`	MM/dd/yyyy	`ShortDatePattern`
`D`	dddd, MMMM dd, yyy	`LongDatePattern`
`f`	dddd, MMMM dd, YYYY HH:mm	Full (long date + short time)
`F`	dddd, MMMM dd, yyyy HH:mm:ss	`FullDateTimePattern` (long date + long time)
`g`	MM/dd/yyyy HH:mm	General (short date + short time)
`G`	MM/dd/yyyy HH:mm:ss	General (short date + long time)
`m`, `M`	MMMM dd	`MonthDayPattern`
`r`, `R`	ddd, dd MMM yy HH':'mm':'ss 'GMT'	`RFC1123Pattern`
`s`	yyyy-MM-dd HH:mm:ss	`SortableDateTimePattern` (ISO 8601)
`S`	YYYY-mm-DD hh:MM:SS GMT	Sortable with time zone information
`t`	HH:mm	`ShortTimePattern`
`T`	HH:mm:ss	`LongTimePattern`
`u`	yyyy-MM-dd HH:mm:ss	Same as `s` but with universal instead of local time
`U`	dddd, MMMM dd, yyyy HH:mm:ss	`UniversalSortableDateTimePattern`

Custom DateTime Format

You can use the patterns listed in Table 34-4 to build a custom format.

Table 34-4. *Custom `DateTime` Format Patterns*

Pattern	Description
`d`	Day of month as digits with no leading zero for single-digit days
`dd`	Day of month as digits with leading zero for single-digit days
`ddd`	Day of week as a three-letter abbreviation
`dddd`	Day of week as its full name
`M`	Month as digits with no leading zero for single-digit months
`MM`	Month as digits with leading zero
`MMM`	Month as three-letter abbreviation
`MMMM`	Month as its full name
`y`	Year as last two digits, with no leading zero
`yy`	Year as last two digits, with leading zero
`yyyy`	Year represented by four digits

The day and month names are determined by the appropriate field in the `DateTimeFormatInfo` class.

Custom Object Formatting

Earlier examples have overridden the `ToString()` function to provide a string representation of a function. An object can supply different formats by defining the `IFormattable` interface and then changing the representation based upon the string of the function.

For example, an `Employee` class could add information with a different format string. This example:

```
using System;
class Employee: IFormattable
{
    public Employee(int id, string firstName, string lastName)
    {
        this.id = id;
        this.firstName = firstName;
        this.lastName = lastName;
    }
```

```
     public string ToString (string format, IFormatProvider fp)
    {
        if ((format != null) && (format.Equals("F")))
            return(String.Format("{0}: {1}, {2}",
                id, lastName, firstName));
        else
            return(id.ToString(format, fp));
    }
    int    id;
    string    firstName;
    string    lastName;
}
class Test
{
    public static void Main()
    {
        Employee fred = new Employee(123, "Fred", "Morthwaite");
        Console.WriteLine("No format: {0}", fred);
        Console.WriteLine("Full format: {0:F}", fred);
    }
}
```

produces the following output:

```
No format: 123
Full format: 123: Morthwaite, Fred
```

The `Format()` function looks for the `F` format. If it finds it, it writes the full information. If it doesn't find it, it uses the default format for the object.

The `Main()` function passes the format flag in the second `WriteLine()` call.

Numeric Parsing

Numbers are parsed using the `Parse()` method provided by the numeric data types. You can pass flags from the `NumberStyles` class to specify which styles are allowed, and you can pass a `NumberFormatInfo` instance to control parsing.

A numeric string produced by any of the standard format specifiers (excluding hexadecimal) is guaranteed to be correctly parsed if the `NumberStyles.Any` style is specified. This example:

```
using System;
class Test
{
    public static void Main()
    {
        int value = Int32.Parse("99953");
        double dval = Double.Parse("1.3433E+35");
        Console.WriteLine("{0}", value);
        Console.WriteLine("{0}", dval);
    }
}
```

produces the following output.

```
99953
1.3433E35
```

Using XML in C#

Although C# supports XML documentation (see Chapter 38), C# doesn't provide any specific language support for using XML.

That's okay, however, because the CLR provides extensive support for XML. Some areas of interest are the `System.Data.Xml` and `System.Xml` namespaces.

Input/Output

The .NET CLR provides I/O functions in the `System.IO` namespace. This namespace contains classes for doing I/O and for other I/O-related functions, such as directory traversal, file watching, and so on.

Reading and writing happens using the `Stream` class, which merely describes how bytes can be read and written to some sort of backing store. `Stream` is an abstract class, so in practice classes derived from `Stream` will be used. The classes listed in Table 34-5 are available.

Table 34-5. *Stream Types in the `System.IO` Namespace*

Class	Description
`FileStream`	A stream on a disk file
`MemoryStream`	A stream that's stored in memory
`NetworkStream`	A stream on a network connection
`BufferedStream`	Implements a buffer on top of another stream
`GZipStream`	A stream that can compress or decompress data, passing through it using GZIP (RFC 1952)
`DeflateStream`	A stream that can compress or decompress data, passing through it using LZW77 (RFC 1951)

With the exception of `BufferedStream`, `GZipStream`, and `DeflateStream`, which sit on top of another stream, each stream defines where the written data will go.

The `Stream` class provides raw functions to read and write at a byte level, both synchronously and asynchronously. Usually, however, it's nice to have a higher-level interface on top of a stream, and you can select from several supplied ones depending on what final format you want.

Binary

The `BinaryReader` and `BinaryWriter` classes read and write values in binary (or raw) format. For example, a `BinaryWriter` can write an `int`, followed by a `float`, followed by another `int`. These classes operate on a stream.

Text

The `TextReader` and `TextWriter` abstract classes define how text is read and written. They allow operations on characters, lines, blocks, and so on. Two implementations of `TextReader` are available.

The somewhat strangely named `StreamWriter` class is the one used for "normal" I/O (opening a file and reading the lines out) and operates on a `Stream`.

The `StringReader` and `StringWriter` classes can read and write from a string.

XML

The `XmlTextReader` and `XmlTextWriter` classes read and write XML. They're similar to `TextReader` and `TextWriter` in design, but they don't derive from those classes because they deal with XML entities rather than text. They're low-level classes used to create or decode XML from scratch.

Serial Ports

New in the .NET Framework 2.0 is support for serial ports. The types that implement serial port support are contained in the `System.IO.Ports` namespace, and they support operations that were previously available only through interop. The main type is `SerialPort`, which represents a physical serial port and allows various properties such as baud rate, parity, and timeouts to be set. `SerialPort` has methods that provide direct access to the data that's flowing through the port and also supports stream-based access so you can use helper streams such as `BufferedStream` or asynchronous operations.

This sample shows both the direct and the stream-based approach:

```
using System.IO.Ports;
…
byte[] buffer = new byte[256];
using (SerialPort sp = new SerialPort("COM1", 19200))
{
 sp.Open();
 //read directly
 sp.Read(buffer, 0, (int)buffer.Length);
 //read using a Stream
 sp.BaseStream.Read(buffer, 0, (int)buffer.Length);
}
```

Reading and Writing Files

You have two ways to get streams that connect to files. The first is to use the `FileStream` class, which provides full control over file access, including access mode, sharing, and buffering:

```
using System;
using System.IO;

class Test
{
    public static void Main()
    {
        FileStream f = new FileStream("output.txt", FileMode.Create);
        StreamWriter s = new StreamWriter(f);

        s.WriteLine("{0} {1}", "test", 55);
        s.Close();
        f.Close();
    }
}
```

The second way is to use the functions in the `File` class to get a stream to a file. This is most useful if a `File` object with the file information is already available, as in the `PrintFile()` function in the next example.

Traversing Directories

This example shows how to traverse a directory structure. It defines both a `DirectoryWalker` class that takes delegates to be called for each directory and file and a path to traverse:

```
using System;
using System.IO;

public class DirectoryWalker
{
public delegate void ProcessDirCallback(DirectoryInfo dir,
  int level, object obj);
    public delegate void ProcessFileCallback(FileInfo file, int level, object obj);

    public DirectoryWalker(    ProcessDirCallback dirCallback,
                ProcessFileCallback fileCallback)
    {
        this.dirCallback = dirCallback;
        this.fileCallback = fileCallback;
    }
```

```
    public void Walk(string rootDir, object obj)
    {
        DoWalk(new DirectoryInfo(rootDir), 0, obj);
    }
    void DoWalk(DirectoryInfo dir, int level, object obj)
    {
        foreach (FileInfo f in dir.GetFiles())
        {
            if (fileCallback != null)
                fileCallback(f, level, obj);
        }
        foreach (DirectoryInfo d in dir.GetDirectories())
        {
            if (dirCallback != null)
                dirCallback(d, level, obj);
            DoWalk(d, level + 1, obj);
        }
    }

    ProcessDirCallback    dirCallback;
    ProcessFileCallback    fileCallback;
}

class Test
{
    public static void PrintDir(DirectoryInfo d, int level, object obj)
    {
        WriteSpaces(level * 2);
        Console.WriteLine("Dir: {0}", d.FullName);
    }
    public static void PrintFile(FileInfo f, int level, object obj)
    {
        WriteSpaces(level * 2);
        Console.WriteLine("File: {0}", f.FullName);
    }
    public static void WriteSpaces(int spaces)
    {
        for (int i = 0; i < spaces; i++)
            Console.Write(" ");

    }
    public static void Main(string[] args)
    {
        DirectoryWalker dw = new DirectoryWalker(
            new DirectoryWalker.ProcessDirCallback(PrintDir),
            new DirectoryWalker.ProcessFileCallback(PrintFile));
```

```
        string root = ".";
        if (args.Length == 1)
            root = args[0];
        dw.Walk(root, "Passed string object");
    }
}
```

Starting Processes

The .NET Framework provides the `Process` class, which starts processes. The following example shows how to start Notepad:

```
// file=process.cs
// compile with csc process.cs
using System.Diagnostics;
class Test
{
    public static void Main()
    {
        ProcessStartInfo startInfo = new ProcessStartInfo();
        startInfo.FileName = "notepad.exe";
        startInfo.Arguments = "process.cs";

        Process.Start(startInfo);
    }
}
```

The arguments used in starting the process are contained in the `ProcessStartInfo` object.

Redirecting Process Output

Sometimes it's useful to get the output from a process. You can do this in the following way:

```
using System;
using System.Diagnostics;
class Test
{
    public static void Main()
    {
        Process p = new Process();
        p.StartInfo.FileName = "cmd.exe";
        p.StartInfo.Arguments = "/c dir *.cs";
        p.StartInfo.UseShellExecute = false;
        p.StartInfo.RedirectStandardOutput = true;
        p.Start();

        string output = p.StandardOutput.ReadToEnd();

        Console.WriteLine("Output:");
        Console.WriteLine(output);    }
}
```

Detecting Process Completion

It's also possible to detect when a process exits:

```
// file=process3.cs
// compile with csc process3.cs
using System;
using System.Diagnostics;
class Test
{
    static void ProcessDone(object sender, EventArgs e)
    {
        Console.WriteLine("Process Exited");
    }

    public static void Main()
    {
        Process p = new Process();
        p.StartInfo.FileName = "notepad.exe";
        p.StartInfo.Arguments = "process3.cs";
        p.EnableRaisingEvents = true;
        p.Exited += new EventHandler(ProcessDone);
        p.Start();
        p.WaitForExit();
        Console.WriteLine("Back from WaitForExit()");
    }
}
```

This example shows two ways of detecting process completion. The `ProcessDone()` function is called when the `Exited` event is fired, and the `WaitForExit()` function also returns when the process is done.

Serialization

Serialization is the process used by the runtime to persist objects in some sort of storage or to transfer them from one location to another.

The metadata information on an object contains sufficient information for the runtime to serialize the fields, but the runtime needs a little help to do the right thing.

Two attributes provide this help. The `[Serializable]` attribute marks an object as okay to serialize. You can apply the `[NonSerialized]` attribute to a field or property to indicate that it shouldn't be serialized. This is useful if it's a cache or derived value.

The following example has a container class named `MyRow` that has elements of the `MyElement` class. The `cacheValue` field in `MyElement` is marked with the `[NonSerialized]` attribute to prevent it from being serialized.

In this example, the `MyRow` object is serialized and deserialized to a binary format and then to an XML format:

```
// file=serial.cs
// compile with: csc serial.cs
using System;
using System.IO;
using System.Collections;
using System.Runtime.Serialization;
using System.Runtime.Serialization.Formatters.Binary;
using System.Runtime.Serialization.Formatters.Soap;

[Serializable]
public class MyElement
{
    public MyElement(string name)
    {
        this.name = name;
        this.cacheValue = 15;
    }
    public override string ToString()
    {
        return(String.Format("{0}: {1}", name, cacheValue));
    }
    string name;
        // this field isn't persisted.
    [NonSerialized]
    int cacheValue;
}
[Serializable]
public class MyRow
{
    public void Add(MyElement my)
    {
        row.Add(my);
    }

    public override string ToString()
    {
        string temp = null;
        foreach (MyElement my in row)
            temp += my.ToString() + "\n";
        return(temp);
    }
```

```
    ArrayList row = new ArrayList();
}

class Test
{
    public static void Main()
    {
        MyRow row = new MyRow();
        row.Add(new MyElement("Gumby"));
        row.Add(new MyElement("Pokey"));

        Console.WriteLine("Initial value");
        Console.WriteLine("{0}", row);

            // write to binary, read it back
        Stream streamWrite = File.Create("MyRow.bin");
        BinaryFormatter binaryWrite = new BinaryFormatter();
        binaryWrite.Serialize(streamWrite, row);
        streamWrite.Close();

        Stream streamRead = File.OpenRead("MyRow.bin");
        BinaryFormatter binaryRead = new BinaryFormatter();
        MyRow rowBinary = (MyRow) binaryRead.Deserialize(streamRead);
        streamRead.Close();

        Console.WriteLine("Values after binary serialization");
        Console.WriteLine("{0}", rowBinary);

            // write to SOAP (XML), read it back
        streamWrite = File.Create("MyRow.xml");
        SoapFormatter soapWrite = new SoapFormatter();
        soapWrite.Serialize(streamWrite, row);
        streamWrite.Close();

        streamRead = File.OpenRead("MyRow.xml");
        SoapFormatter soapRead = new SoapFormatter();
        MyRow rowSoap = (MyRow) soapRead.Deserialize(streamRead);
        streamRead.Close();

        Console.WriteLine("Values after SOAP serialization");
        Console.WriteLine("{0}", rowSoap);
    }
}
```

The example produces the following output:

```
Initial value
Gumby: 15
Pokey: 15

Values after binary serialization
Gumby: 0
Pokey: 0

Values after SOAP serialization
Gumby: 0
Pokey: 0
```

The field `cacheValue` isn't preserved, since it was marked as `[NonSerialized]`. The file `MyRow.Bin` will contain the binary serialization, and the file `MyRow.xml` will contain the XML version.

The XML encoding is in SOAP, which is a textual dump of the objects, in an (almost) human-readable form. To produce a specific XML encoding, use the `XmlSerializer` class.

Custom Serialization

If the standard serialization doesn't do exactly what you want or doesn't give sufficient control, a class can define exactly how it wants to be serialized,[1] such as in this example:

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

class Employee: ISerializable
{
    int id;
    string name;
    string address;

    public Employee(int id, string name, string address)
    {
        this.id = id;
        this.name = name;
        this.address = address;
    }
```

1. If you're familiar with how MFC serialization worked in Visual C++, this approach will seem fairly familiar.

```
    public override string ToString()
    {
        return(String.Format("{0} {1} {2}", id, name, address));
    }

    Employee(SerializationInfo info, StreamingContext content)
    {
        id = info.GetInt32("id");
        name = info.GetString("name");
        address = info.GetString("address");
    }

        // called to save the object data...
    public void GetObjectData(SerializationInfo info, StreamingContext content)
    {
        info.AddValue("id", id);
        info.AddValue("name", name);
        info.AddValue("address", address);
    }
}

class Test
{
    public static void Serialize(Employee employee, string filename)
    {
        Stream streamWrite = File.Create(filename);
        IFormatter writer = new SoapFormatter();
        writer.Serialize(streamWrite, employee);
        streamWrite.Close();
    }

    public static Employee Deserialize(string filename)
    {
        Stream streamRead = File.OpenRead(filename);
        IFormatter reader = new SoapFormatter();
        Employee employee = (Employee) reader.Deserialize(streamRead);
        streamRead.Close();
        return(employee);
    }

    public static void Main()
    {
        Employee employee = new Employee(15, "Fred", "Bedrock");
```

```
        Serialize(employee, "emp.dat");
        employee = Deserialize("emp.dat");
        Console.WriteLine("Employee: {0}", employee);
    }
}
```

To do customer serialization, an object must implement the `ISerializable` interface. The `GetObjectData()` method is the only method on that interface. The implementation of that method stores each value by calling `AddValue()` on each value and passing in a name for the field and the field value.

To deserialize an object, the runtime relies on a special constructor. This constructor will call the appropriate `get` function to fetch a value based on the name.

Although this approach takes some extra space to store the names—and a bit of time to look them up—it versions well, allowing new values to be added without invalidating existing stored files.

Reading Web Pages

The following example demonstrates how to write a "screen scraper" using C#. The following bit of code will take a stock symbol, format a URL to fetch a quote from the MSN Money site, and then extract the quote from the HTML page using a regular expression:

```
using System;
using System.Net;
using System.IO;
using System.Text;
using System.Text.RegularExpressions;

class QuoteFetch
{
    public QuoteFetch(string symbol)
    {
        this.symbol = symbol;
    }

    public string Last
    {
        get
        {
            string url = "http://moneycentral.msn.com/scripts/" +
              "webquote.dll?ipage=qd&Symbol=";
            url += symbol;

            ExtractQuote(ReadUrl(url));
            return(last);
        }
    }
```

```
    string ReadUrl(string url)
    {
        Uri uri = new Uri(url);

            //Create the request object

        WebRequest req = WebRequest.Create(uri);
        WebResponse resp = req.GetResponse();
        Stream stream = resp.GetResponseStream();
        StreamReader sr = new StreamReader(stream);

        string s = sr.ReadToEnd();

        return(s);

    }
    void ExtractQuote(string s)
    {
        // Line like: "Last</TD><TD ALIGN=RIGHT NOWRAP><B> 78 3/16"

        Regex lastmatch = new Regex(@"Last\D+(?<last>.+)<\/B>");
        last = lastmatch.Match(s).Groups[1].ToString();
    }
    string    symbol;
    string    last;
}

class Test
{
    public static void Main(string[] args)
    {
        if (args.Length != 1)
            Console.WriteLine("Quote <symbol>");
        else
        {
            // GlobalProxySelection.Select = new DefaultControlObject("proxy", 80);
            QuoteFetch q = new QuoteFetch(args[0]);
            Console.WriteLine("{0} = {1}", args[0], q.Last);
        }
    }
}
```

In this age of Web services, screen scrapers are generally seen as passé. If a Web service is available that offers the equivalent functionality, you should use it over a screen scraper. However, it's interesting to note that it has been more than three years since this code was written (in the first edition of this book), and it still works as well today as it did when first written.

■ **Note** When working from behind a firewall, it may be necessary to set a proxy. You can do this with the commented code in `Main()`.

Accessing Environment Settings

You can use the `System.Environment` class to obtain information about the machine and environment, as the following example demonstrates:

```
using System;
using System.Collections;

class Test
{
    public static void Main()
    {
         Console.WriteLine("Command Line: {0}",
          Environment.CommandLine);
         Console.WriteLine("Current Directory: {0}",
          Environment.CurrentDirectory);
         Console.WriteLine("HasShutdownStarted: {0}",
          Environment.HasShutdownStarted);
         Console.WriteLine("Machine Name: {0}",
          Environment.MachineName);
         Console.WriteLine("OS Version: {0}",
          Environment.OSVersion);
         Console.WriteLine("ProcessorCount: {0}",
          Environment.ProcessorCount);
         Console.WriteLine("Stack Trace: {0}",
          Environment.StackTrace);
         Console.WriteLine("System Directory: {0}",
          Environment.SystemDirectory);
         Console.WriteLine("Tick Count: {0}",
          Environment.TickCount);
         Console.WriteLine("Version: {0}", Environment.Version);
         Console.WriteLine("UserDomainName: {0}",
          Environment.UserDomainName);
         Console.WriteLine("UserInteractive: {0}",
          Environment.UserInteractive);
         Console.WriteLine("UserName: {0}", Environment.UserName);
         Console.WriteLine("Working Set: {0}",
          Environment.WorkingSet);
```

```
        Console.WriteLine("Environment Variables");
        foreach (DictionaryEntry var in
          Environment.GetEnvironmentVariables())
        Console.WriteLine("    {0}={1}", var.Key, var.Value);

        Console.WriteLine("Logical Drives");
        foreach (string drive in Environment.GetLogicalDrives())
        Console.WriteLine("    {0}", drive);    }
}
```

When this runs, it generates the following output (results will vary on different machines):

```
Command Line: "Z:\Nick\PIC#\Code\Accessing Environment
 Settings\bin\Debug\Accessing Environment Settings.vshost.exe"
Current Directory: Z:\Nick\PIC#\Code\Accessing Environment
 Settings\bin\Debug
HasShutdownStarted: False
Machine Name: WINXP-VMWARE
OS Version: Microsoft Windows NT 5.1.2600 Service Pack 2
ProcessorCount: 1
Stack Trace:    at System.Environment.GetStackTrace(Exception e,
 Boolean needFileInfo)
   at System.Environment.get_StackTrace()
   at Test.Main() in z:\nick\pic#\code\accessing environment
settings\file_1.cs:line 16
   at System.AppDomain.nExecuteAssembly(Assembly assembly,
String[] args)
   at System.AppDomain.ExecuteAssembly(String assemblyFile,
 Evidence assemblySecurity, String[] args)
   at Microsoft.VisualStudio.HostingProcess.Utilities.HostProc.
     RunUsersAssembly()
   at System.Threading.ThreadHelper.ThreadStart_Context(Object
 state)
   at System.Threading.ExecutionContext.Run(ExecutionContext
 executionContext, ContextCallback callBack, Object state,
 StackCrawlMark& stackMark)
   at System.Threading.ThreadHelper.ThreadStart()
System Directory: C:\WINDOWS\system32
Tick Count: 10894000
Version: 2.0.41202.0
UserDomainName: WINXP-VMWARE
UserInteractive: True
UserName: User
Working Set: 11493376
```

```
Environment Variables
    Path=C:\WINDOWS\system32;C:\WINDOWS;C:\WINDOWS\System32\Wbem
    TEMP=C:\DOCUME~1\User\LOCALS~1\Temp
    SESSIONNAME=Console
    PATHEXT=.COM;.EXE;.BAT;.CMD;.VBS;.VBE;.JS;.JSE;.WSF;.WSH
    USERDOMAIN=WINXP-VMWARE
    PROCESSOR_ARCHITECTURE=x86
    SystemDrive=C:
    APPDATA=C:\Documents and Settings\User\Application Data
    windir=C:\WINDOWS
    TMP=C:\DOCUME~1\User\LOCALS~1\Temp
    USERPROFILE=C:\Documents and Settings\User
    ProgramFiles=C:\Program Files
    FP_NO_HOST_CHECK=NO
    HOMEPATH=\Documents and Settings\User
    COMPUTERNAME=WINXP-VMWARE
    USERNAME=User
    NUMBER_OF_PROCESSORS=1
    PROCESSOR_IDENTIFIER=x86 Family 15 Model 2 Stepping 8,
     GenuineIntel
    SystemRoot=C:\WINDOWS
    ComSpec=C:\WINDOWS\system32\cmd.exe
    LOGONSERVER=\\WINXP-VMWARE
    VS80COMNTOOLS=C:\Program Files\Microsoft Visual Studio
     8\Common7\Tools\
    WecVersionForRosebud.740=2
    CommonProgramFiles=C:\Program Files\Common Files
    PROCESSOR_LEVEL=15
    PROCESSOR_REVISION=0208
    CLIENTNAME=Console
    ALLUSERSPROFILE=C:\Documents and Settings\All Users
    OS=Windows_NT
    HOMEDRIVE=C:
Logical Drives
    A:\
    C:\
    D:\
    Z:\
```

CHAPTER 35

Windows Forms

You use the Windows Forms section of the .NET Framework to write *rich-client* applications (otherwise known as *Windows applications*).

This chapter covers the initial steps of developing an application in Windows Forms. You'll get the chance to expand the application in later chapters.

Note We've written the chapters that describe this application at the same time as writing the application itself in order to present the process of application development. We've tried to be reasonably honest about any refactoring that was required along the way, and we've tried to explain our rationale for choosing specific approaches.

Creating Your Application

To best understand how to write Windows Forms applications, it's useful to develop a real application.

Recently, one of our test machines was running out of space on its C: drive. To try to find out what was happening, we spent some time looking in likely directories, but it wasn't an easy task. What we really wanted was a way to see what had changed so we'd know exactly where the disk space went.

The solution is an application named DiskDiff. It will show a tree view of the space used on a system, and it can compare the space used at two different time periods.

Getting Started

The first task is to create a default Windows Forms application. You do this by creating a new C# project and choosing Windows Application as the template. The name is DiskDiff, as shown in Figure 35-1.

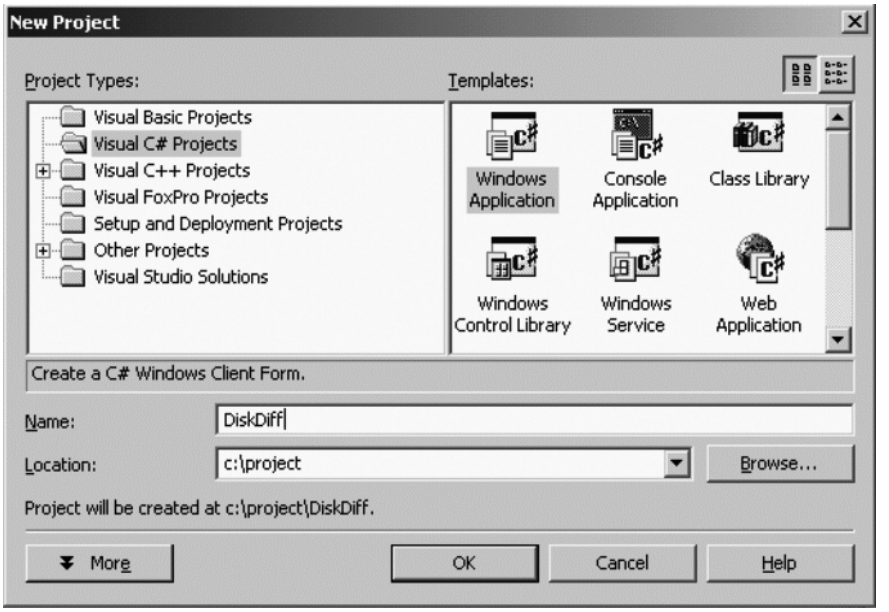

Figure 35-1. *Creating a Windows Application project*

The VS .NET environment will create the initial project and show the initial form. In Windows Forms projects, a form is simply a class derived from `System.Windows.Forms.Form`. You'll see these forms listed under a project in the Solution Explorer with a special form icon, as shown in Figure 35-2.

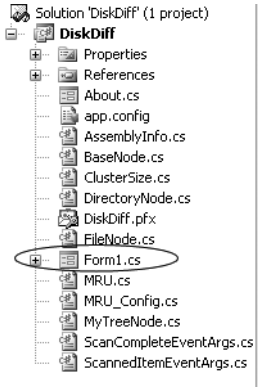

Figure 35-2. *Solution Explorer with Windows Forms icon circled*

You can view a form class in two ways. You can view the code of the form by right-clicking the form and choosing View Code. By double-clicking the form (or by right-clicking and choosing View Designer), you can view the form in the Form Designer (more about that in a bit).

The Windows Forms architecture differs from other Microsoft approaches in that the layout and controls for a form are implemented in the code of the form class, rather than in some sort of resource storage.

The initial class for this project looks like this (with the XML comments removed for clarity):

```
namespace DiskDiff
{
    using System;
    using System.Drawing;
    using System.Collections;
    using System.ComponentModel;
    using System.Windows.Forms;
    using System.Data;

    public class Form1 : System.Windows.Forms.Form
    {
        private System.ComponentModel.Container components;

        public Form1()
        {
            //
            // Required for Windows Forms Designer support
            //
            InitializeComponent();

            //
            // TODO: Add any constructor code after InitializeComponent call
            //
        }

        public override void Dispose()
        {
            base.Dispose();
            components.Dispose();
        }

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

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

This class is fairly simple. The constructor for the class calls `InitializeComponent()`, which is the member that the class designer will use. Initially, all it has is a container for the other items on the form and lines to set the size and text of the form. A `Dispose()` member will clean up the form when it closes.[1]

Finally, the `Main()` function for the whole application creates an instance of the form class and then calls `Application.Run()`, passing the form. This will execute the standard Windows message loop to get messages and dispatch them to the appropriate objects.

Using the Form Designer

The VS .NET environment makes laying out forms easy. You can view the Toolbox by clicking the Toolbox tab along the left side of the development environment. Selecting Windows Forms will show all the controls you can place on a form.

The directories you want to show make the `TreeView` a good choice, so drag one onto your form. The designer will show you the `TreeView` object and add the following code to the `InitializeComponent()` method:

```
this.treeView1 = new System.Windows.Forms.TreeView ();
treeView1.Location = new System.Drawing.Point (200, 112);
treeView1.Size = new System.Drawing.Size (121, 97);
treeView1.TabIndex = 0;
```

You'll want the `TreeView` control to take up all the space in the form. You can do this by resizing the control when the size of the form changes, but the Windows Forms architecture can do this for you. If the `TreeView` object is selected, you can set the `Anchor` property in the property window. Initially, the object is anchored to the top-left corner; by setting the anchor to all four sides, you can resize the tree view whenever the form is resized. We've used anchoring in this case, but since this form has only a single object, you could achieve the same effect by setting the `Dock` property on the `TreeView` to `DockStyle.Fill`.

Now that the basic framework of the application is ready, it's time to move onto some disk operations.

Finding Directory Sizes

The base of DiskDiff is the code that can traverse a directory tree and figure out the sizes of the files there.

A quick look in the `System.IO` namespace shows a `DirectoryInfo` class that will work. You'll encapsulate it in a class of your own so you can store the list of files and the list of directories in each directory. The class will look like this:

1. Forms use system resources that the garbage collector can't track, so you can call the `Dispose()` method to clean these up before the garbage collector gets around to the next collection. See Chapter 8 for more information on `Dispose()`.

```
    public class DirectoryNode
    {
        string root;
        List<FileNode> files = new List<FileNode>();
        List<DirectoryNode> dirs = new List<DirectoryNode>();
        DirectoryInfo directoryInfo;    // this directory

        public DirectoryNode(string root)
        {
            this.root = root;
            directoryInfo = new DirectoryInfo(root);
        }
        public DirectoryNode(DirectoryInfo directoryInfo)
        {
            this.directory = directoryInfo;
            this.root = directoryInfo.FullName;
        }
        public void Populate()
        {
            foreach (FileInfo f in directoryInfo.GetFiles())
            {
                FileNode fileNode = new FileNode(f);
                this.files.Add(f);
            }

            foreach (DirectoryInfo d in directory.GetDirectories())
            {
                DirectoryNode dirNode = new DirectoryNode(d.FullName);
                dirs.Add(dirNode);
                dirNode.Populate();
            }
        }
        public void PrintTree(int level)
        {
            for (int i = 0; i < level; i++)
            {
                Console.Write("  ");
            }
            Console.WriteLine("{0}", this.root);
            foreach (DirectoryNode dirNode in dirs)
            {
                dirNode.PrintTree(level + 1);
            }
        }
}
```

`DirectoryNode` has two constructors. The first creates the top-level directory, and the other creates a `DirectoryNode` from a `DirectoryInfo` object.

The `Populate()` function is the heart of the class. It uses the `DirectoryInfo` object that's encapsulated, calling `GetFiles()` to get the list of files in the directory and `GetDirectories()` to get the list of directories. It then recurses for each subdirectory, so the directory tree can be fully traversed.

The `PrintTree()` function is used for testing, along with a little test program:

```
class Test
{
    public static void Main()
    {
        DirectoryNode directoryNode = new DirectoryNode(@"c:\project\diskdiff");
        directoryNode.Populate();
        directoryNode.PrintTree(0);
    }
}
```

On our system, this gives the following output:

```
c:\project\diskdiff
  c:\project\diskdiff\bin
    c:\project\diskdiff\bin\Debug
  c:\project\diskdiff\obj
    c:\project\diskdiff\obj\Debug
      c:\project\diskdiff\obj\Debug\temp
      c:\project\diskdiff\obj\Debug\TempPE
```

Calculating Sizes

Now that you have the directory code, you need to sum the sizes of the files for a directory and then pass that size up to the parent directories. You'll want to be able to fetch both the size of this directory and the size of this directory and the subdirectories under it. For this, you'll add two properties:

```
long? size = null;            // size of dir in bytes
long? sizeTree = null;        // size of dir and subdirs
public long Size
{
    get
    {
        if (size == null)
        {
            size = 0;
```

```
            foreach (FileNode f in files)
            {
                size += f.fileInfo.Length;
            }
        }
        return(size);
    }
}
public long SizeTree
{
    get
    {
        if (sizeTree == null)
        {
            sizeTree = 0;
            sizeTree += Size;
            foreach (DirectoryNode dirNode in dirs)
            {
                sizeTree += dirNode.SizeTree;
            }
        }
        return(sizeTree);
    }
}
```

The `Size` property simply walks through all the files in the current node and adds their sizes. This total size is then stored in the `size` variable so it doesn't need to be recalculated.

The `SizeTree` property adds the size of all the subdirectories to the current size. Because `SizeTree` recourses down the tree, getting the value of the property at the root will cause the sizes to be calculated all the way down the tree.

Although this is a nice way to use properties, it may turn out that calculating the sizes during the call to `Populate()` is a better choice.

The `PrintTree()` is renamed to `PrintSizes()` and now prints out the values of `Tree` and `TreeSize` next to the name of the directory. Running the code produces the following output:

```
c:\project\diskdiff 32632 119672
  c:\project\diskdiff\bin 0 43520
    c:\project\diskdiff\bin\Debug 43520 43520
  c:\project\diskdiff\obj 0 43520
    c:\project\diskdiff\obj\Debug 43520 43520
      c:\project\diskdiff\obj\Debug\temp 0 0
      c:\project\diskdiff\obj\Debug\TempPE 0 0
```

Now it's time to integrate the directory traversal code into the `TreeView` class.

A Debugging Suggestion

Though a separate `DirectoryNode.cs` file is part of the example, the class was written in the main project. For testing purposes, we wanted to print the traversal of a big directory, but since `DiskDiff` is a Windows Forms project, it doesn't have a console window.

However, a Windows Forms project can have a console window. By right-clicking the project in the Solution Explorer and choosing Properties, you can change the output type of the project to `Console Application`.

This means you'll have to dismiss the console window when exiting the application, but that's fine for debugging. When you're finished, just change the output type of the project back, and the console window will go away.

Displaying the Directory Tree and Sizes

You must now decide how to interface the form and the `DirectoryNode` classes so the tree can be populated. One option is to have `DirectoryNode` expose enough of its internals (perhaps through an indexer) so the form can iterate over it. Another option is to pass the `TreeView` control to a function in `DirectoryNode` and have it populate the control.

A trade-off exists between having `DirectoryNode` expose a more complex interface and having `DirectoryNode` be more tightly coupled to the form. In this case, we'll choose the first option, but the second option can sometimes be better, especially if the processing to be done is complex.

Because a directory can be thought of as an array of files, an indexer is a reasonable choice. However, you need to differentiate between directories and files, so doing what the `Directory` class does, with `GetFiles()` and `GetDirectories()` members, is a better choice. Rather than return the collection held by `DirectoryNode`, a copy of the collection is made, sorted, and returned. This allows the UI to make any modifications it needs to the collection without affecting other objects that might be using the same `DirectoryNode` instance.

The code for these functions is as follows:

```
public DirectoryNode[] GetDirectories()
{
        DirectoryNode[] array = dirs.ToArray();
        Array.Sort(array);
        return (array);
}

public FileNode[] GetFiles()
{
        FileNode[] array = files.ToArray();
        Array.Sort(array);
        return (array);
}
```

This is a good example of how garbage collection can simplify interaction between different objects; in C++, you'd have to carefully consider the question of who owned the objects in the returned array.

The code to populate the `TreeView` object is only a few lines:

```
public void PopulateTree(TreeNodeCollection treeNodeCollection,
DirectoryNode directoryNode)
{
    TreeNode treeNode = new TreeNode(directoryNode.NameSize);
    treeNodeCollection.Add(treeNode);
    foreach (DirectoryNode subdir in directoryNode.GetDirectories())
    {
        PopulateTree(treeNode.Nodes, subdir);
    }

    foreach (FileNode fileNode in directoryNode.GetFiles())
    {
        TreeNode treeFileNode = new TreeNode(fileNode.NameSize);
        treeNode.Nodes.Add(treeFileNode);
    }
}
```

This is a recursive function. It adds a `TreeNode` instance to the collection for the current level and then recurses to add all the subdirectories under this directory as children of this directory.

It then adds entries for all the `FileNode` objects in this directory. The `FileNode` object was added to encapsulate functions dealing with files—primarily the `NameSize` property that returns the properly formatted string for this file.

When this code is run, it generates the window (after expanding all the nodes) shown in Figure 35-3.

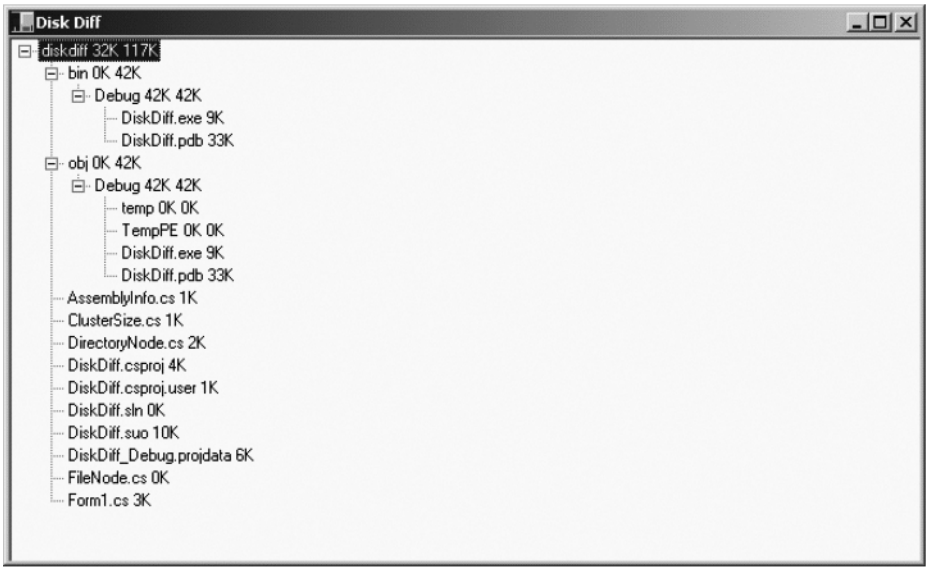

Figure 35-3. *DiskDiff view of directory tree*

Setting the Directory

The `FolderBrowserDialog` class allows a directory to be selected in a standard Windows manner, as shown in Figure 35-4.

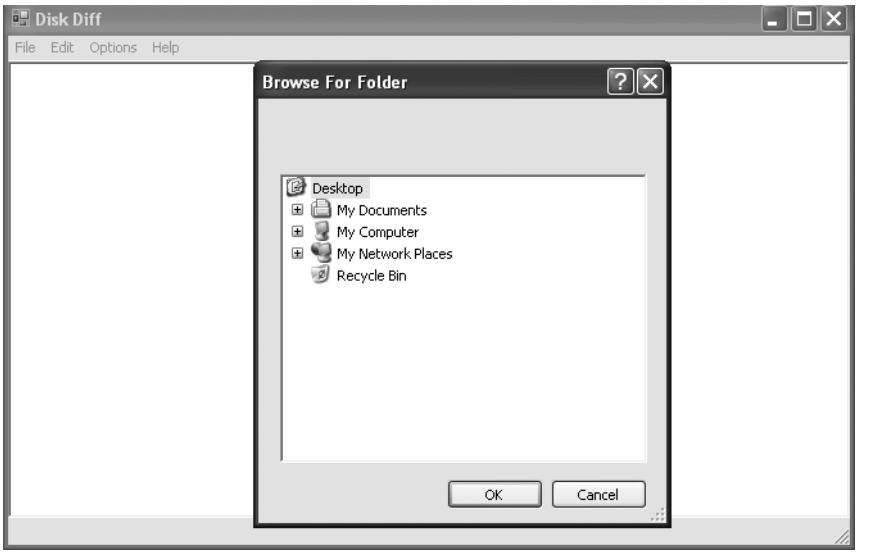

Figure 35-4. *Browsing for a directory to scan*

The following is the code to display and use `FolderBrowserDialog`:

```
FolderBrowserDialog dialog = new FolderBrowserDialog();
dialog.ShowNewFolderButton = false;
if (dialog.ShowDialog() == DialogResult.OK)
{
    directoryNodeBaseline = null;
    rootDirectory = dialog.SelectedPath;
    DoTree();
}
```

This code has a slight problem, however. If the root directory `c:\` is entered, it will take the code quite a while to fetch all the information, and the program will appear to hang. That's bad.

Tracking Your Progress

The first step is to indicate to the user that something is happening. One option is to add a `ProgressBar` control to the program. The problem with doing this is that you don't know how many files you'll need to process, so it's difficult to know what the endpoint of the process should mean.

This means that either the progress bar will complete many times or it will never complete, which may be misleading to the user.

Another option is to indicate some directory or file information in the status bar. To start, drag a `StatusBar` control from the Toolbox to the form, and set the initial text of the status bar to an empty string so there isn't any text displayed to start.

The next thing to do is to figure out how to update the `StatusBar` control when the directories are being scanned. You can do this in two ways. The first is to change the `DirectoryNode.Populate()` member so it takes a `StatusBar` as a parameter and have it change the text in the `StatusBar` directly.

That works fine, but it means the `DirectoryNode` class needs to know how the program wants to handle updating information. It's not a very general implementation.

The second option is to have the `DirectoryNode` class support an event that's fired whenever a file or directory is scanned. Since that's a bit nicer, that's what we'll show how to do for this example. A downside of this approach is that you'll need two event fields in each `DirectoryNode` object, even though only the root will ever have events hooked to it.[2]

The first step is to define a class that will carry the event information:

```
public class ScannedItemEventArgs: EventArgs
      {
          string name;
           public ScannedItemEventArgs(string name)
           {
               this.name = name;
           }

           public string Name
           {
               get
               {
                   return name;
               }
           }
       }
```

By convention, `EventArgs` is used as a base class for any class passing information for an event.

The next step is to add a delegate to `DirectoryNode`:

```
public delegate void ScannedEventHandler(object sender, ScannedItemEventArgs e);
```

Then, you'll declare two events: one for a directory being scanned and one for a file being scanned:

```
public event ScannedEventHandler DirectoryScanned;
public event ScannedEventHandler FileScanned;
```

Finally, you'll add functions to fire the events. Having separate functions for these isn't strictly necessary, but they make the class a bit easier to use and easier to derive from.

The event framework is now ready. The next step is to modify the code in `Populate()` so it fires the events. An interesting subtlety exists here. The class that uses the events will hook up

2. If necessary, you can overcome this overhead by using the advanced event syntax.

to the events declared in the root directory node, but since the `Populate()` function is recursive, it can't fire the events using its own events. Instead, it needs to use the root's events.

You do this by writing a private version of `Populate()` that takes a `DirectoryNode` as a parameter and by passing the root object as a parameter during recursive calls. You then modify the public version to call the private version.

To hook up the events, you need to declare the functions to be called when the event is fired, as part of the main form class:

```
void myFileScanned(object sender, ScannedItemEventArgs e)
{
    statusBar1.Text = "File: " + e.Name;
}

void myDirectoryScanned(object sender, ScannedItemEventArgs e)
{
    statusBar1.Text = "Directory: " + e.Name;
}
```

These functions merely update the status bar with the name of the file. You'll also need to hook those to the events in the `DirectoryNode` object. Here's the relevant code:

```
try
{
                DirectoryNode newNode = new DirectoryNode(rootDirectory);

        directoryNode = newNode;
        directoryNode.DirectoryScanned +=
                        new DirectoryNode.ScannedEventHandler(myDirectoryScanned);
                directoryNode.FileScanned +=
                        new DirectoryNode.ScannedEventHandler(myFileScanned);
                directoryNode.Populate(null);

                treeView1.Nodes.Clear();
                PopulateTree(treeView1.Nodes, directoryNode);
            }
            catch (DirectoryNotFoundException e)
            {
                // don't do anything; happens on some system directories.
            }
```

The `try` block in this code handles an invalid entry for the root directory; in that case, the `catch` block will execute, and the tree will still be valid.

After the new `DirectoryNode` instance is created, the event handlers are added to it, the directory is populated, and then the tree view is refreshed with the new data.

When this code runs, it will update the status bar as it walks through the specified directory tree. It's a bit distracting to see an update on every file (not to mention a bit wasteful to fire an event for every file), so you'll need to figure out a way to update less often. Our initial idea was to merely update the status bar for directories, but this meant there was no change for a

few seconds with really big directories and too many updates for small directories. What you need is an update based on time rather than on the item that's being scanned.

Add two items to the `DirectoryNode` class:

```
TimeSpan scanUpdateMinimum = new TimeSpan(0, 0, 0, 0, 200);
DateTime lastUpdateTime;
```

The `scanUpdateMinimum` field defines the time period for which there should be no update. The constructor call sets this period to 200 milliseconds. The `lastUpdateTime` field stores the time of the last update.

Then add two lines of code to the event-firing functions `OnDirectoryScanned` and `OnFileScanned`:

```
if ((DateTime.Now - this.lastUpdateTime) < scanUpdateMinimum)
    return;
```

This code calculates how long it has been since the last update was sent and doesn't send the update if there was a recent update.

It's now obvious that the application isn't hanging, but there's no way to interrupt the application in the middle of a scan. You'll tackle that in Chapter 38.

CHAPTER 36

DiskDiff: More Sophistication

Now that the basic outline of your DiskDiff application is done, it's time to make it better.

Populating on a Thread

To make your application behave, you need to do the scan on a different thread so the user-interface thread can continue operating. In this example, you'll use the `Thread` object from the `System.Threading` namespace. Starting the thread is easy:

```
public void Populate()
{
    Thread t = new Thread(new ThreadStart(DoPopulate));
    t.Start();
}
```

The function that will be called at the start of the thread is `DoPopulate()`. To create a new thread, a `ThreadStart` delegate must be created on the function you want called and passed to the thread. Then, the `Start()` member on the thread is called, and the thread starts and runs on its merry way.

That gets the process working, but your app is now broken. When the `DoTree()` function in the form calls `Populate()`, it will start the thread and return immediately and then try to repaint the tree form. This is bad, because the information isn't ready to paint yet.

To fix this, you'll add a new event to the `DirectoryNode` object for when the populate function is done:

```
void DoPopulate()
{
    DoPopulate(this);
    OnPopulateComplete();
}
```

Because the delegate method doesn't have much code, you can convert it to an anonymous method and place it with the thread's creation:

```
public void Populate()
{
    cancelled = false;
    Thread t = new Thread(delegate() {
        DoPopulate(this);
        OnPopulateComplete(true);
    });
    t.Start();
}
```

Finally, add a function to the form to repaint the tree when the population is done:

```
void DoTreeDone(object sender, EventArgs e)
{
    Console.WriteLine("DoTreeDone");
    statusBar1.Text = "";

    treeView1.Nodes.Clear();

    PopulateTree(treeView1.Nodes, directoryNode);
}
```

And, to finish, you'll hook this function up to the event.

Now, the user interface is still active while the population is happening, so you can move the application around and have it paint correctly.

At least, that's what you hope will happen, but you have a little problem. The thread doing the population isn't allowed to do anything that updates the control, and adding a node to the node collection automatically updates the tree view. When you try to do this, the system throws an exception that tells you the update can't be done on the current thread.

Very nicely, however, the exception that's thrown tells you exactly what to do; you need to use `Control.Invoke()` to pass a delegate to the function you want to call, and `Control.Invoke()` will find the control and arrange to have the function called on the proper thread.

This is actually pretty easy to set up. The first step is to declare a delegate to the function you want to call:

```
delegate int AddDelegate(TreeNode treeNode);
```

This delegate matches the signature of the function you want to call. The next step is to modify the call. Instead of this:

```
treeNodeCollection.Add(treeNode);
```

you need the following:

```
AddDelegate addDelegate = new AddDelegate(treeNodeCollection.Add);
treeView1.Invoke(addDelegate, new object[] {treeNode});
```

The first line sets up the pointer to the function, and the second one passes it off to the control to be called, along with an array of parameters to pass to the function.

With that change, the program starts working again, but if you point it at a big drive, it might take a long time to complete, and you have no way to interrupt it.

Interrupting a Thread

You'll modify the `DirectoryNode` class to add a `CancelPopulate()` member function. This function will set an internal flag that the populate code will poll, and the code will terminate if it finds that the flag is set. Because the flag is being modified without thread locking, the `volatile` keyword has been applied to the flag to let the runtime know it should check for the most current value of this variable rather than using a copy cached on a particular thread.

When doing polling, how often you poll usually comes with a trade-off. If the polling takes place too often, it can take up more time than the processing. Conversely, if the polling takes place rarely, it can make the cancel appear to have no effect. In this case, the polling takes place before processing each file in a directory.

Another option is for the `DirectoryNode` to store the thread instance and then stop the thread directly by calling the `Abort()` method. This has less overhead since there's no polling, but it makes the code a bit less obvious,[1] and it also could make cleanup a bit more complex since the object could be left in a bad state when the thread stops. To properly do the cleanup, the processing loop wants to catch the `ThreadAbortException` that will be thrown when `Abort()` is called.

Modifying `Populate()` is simple; the file-processing loop simply tests the flag and aborts the thread if the flag is set:

```
foreach (FileInfo f in directory.GetFiles())
{
    if (rootDirectoryNode.cancelled)
    {
        Thread.CurrentThread.Abort();
    }
    rootDirectoryNode.OnFileScanned(f.Name);
    this.files.Add(new FileNode(f));
}
```

This will interrupt the processing, but unfortunately the user interface never finds out the operation was interrupted,[2] so it doesn't have the opportunity to clear the status bar. You can easily fix this by slightly modifying the `PopulateComplete` event so the delegate takes a success parameter (so the user interface can do different things in the success and cancel cases), and the call when the processing is canceled becomes the following:

```
rootDirectoryNode.OnPopulateComplete(false);
Thread.CurrentThread.Abort();
```

1. Polling makes it clear that an operation can be interrupted.
2. The user interface may have generated the event, but it could also have come from somewhere else.

A Cancel Button

To keep things simple, this example will use a button to cancel the processing. Using the designer, you can add a Cancel button at the lower-left corner of the form on top of the tree view. So that it isn't in the way, set the visibility to false.

When processing is started, the visibility is set to true, and the button is now visible and selectable. The event handler that's associated with completing the population will set the visibility back to false when the processing is completed or aborted.

Figure 36-1 shows a view of the application while processing a big directory tree.

Figure 36-1. *DiskDiff with a Cancel button*

You now have an application that can be interrupted. In this case, using threads was a simple way to get what you wanted. Another way to do this is to use the asynchronous call mechanism in the .NET CLR (see Chapter 31).

Decorating the TreeView

The application displays the sizes well, but it's fairly difficult to tell where the disk space is used based upon the numbers. What you need is a graphical representation of the amount of space used by each directory and file.

The `TreeView` control can put bitmaps in front of each item. To do this, an `ImageList` that contains all the bitmaps is attached to the `TreeView`, and then the index of the desired bitmap is set for each item in the tree.

To get started, you need to add an `ImageList` to the form and then populate it with images. First, drag an `ImageList` object from the Toolbox to the form. You can then add images through the `Collections` property in the property window. (We drew the images for this project using Paint Shop Pro.)

To hook up the `ImageList` to the `TreeView`, set the `ImageList` property in the `TreeView` control to the name of the `ImageList` in the form.

The next task is to figure out which image to present for each item in the list. The bitmaps are pie charts representing the percentage of the total space an item used. You can therefore determine the index by multiplying the fraction of space used by the number of images (8) to get the index. That requires you to modify the population function as follows:

```
public void PopulateTree(TreeNodeCollection treeNodeCollection,
                                        DirectoryNode directoryNode,
                                        float fractionUsed)
{
    TreeNode treeNode = new TreeNode(directoryNode.NameSize);
    treeNode.ImageIndex = FractionToIndex(fractionUsed);
    treeNodeCollection.Add(treeNode);

        // As we walk though the tree, we need to figure out the
        // percentages for each item. We do that based upon the
        // full size of this directory.
    float dirSize = directoryNode.SizeTree;
    foreach (DirectoryNode subdir in directoryNode.GetDirectories())
    {
        PopulateTree(treeNode.Nodes, subdir, subdir.SizeTree / dirSize);
    }

    foreach (FileNode fileNode in directoryNode.GetFiles())
    {
        TreeNode treeFileNode = new TreeNode(fileNode.NameSize);
        treeFileNode.ImageIndex =
                            FractionToIndex(fileNode.Size / dirSize);
        treeNode.Nodes.Add(treeFileNode);
    }
}
```

The percentage is passed into the `PopulateTree()` function because each directory creates its own node. The top node is passed in the value `1.0`, since it obviously contains all the size in the tree. The fraction of each element is computed based on the size of that element and the size of the directory it's in, and the index for that fraction is obtained from the `FractionToIndex()` function.

Figure 36-2 shows how a tree looks with all the nodes expanded.

The program is starting to get useful, but it still has a few problems. One is that populating the `TreeView` object with all the nodes of the directory tree can take a long time—and use lots of memory—if the directory tree is large.

Figure 36-2. *TreeView with size icons*

Expand-o-Matic

Instead of populating the whole tree, you'll populate only the currently visible portion of the tree initially. When a user clicks the plus sign to expand a directory, you'll populate the newly visible section.

Hooking up to the `BeforeExpand` event is fairly easy. When this event occurs, a `TreeViewCancelEventArgs` is passed, and that class contains the node that's expanding. The event code will merely need to figure out what `DirectoryNode` object corresponds to the node passed with the event.

When dealing with a tree view control in the MFC framework or directly in Win32, you can store a value with each tree node and use this value to point to the object that corresponds to the tree node.

However, WinForms doesn't provide access to this, so you'll need to use another approach. One approach is to change `DirectoryNode` and `FileNode` so they're derived from `TreeNode` and then store those directly. This works fine but will complicate those classes, since they now depend on the WinForms classes.

Another approach is to define a class derived from `TreeNode` that has a reference to the `DirectoryNode` or `FileNode` object for that node. That class is simple and looks like this:

```
public class MyTreeNode: TreeNode
{
    object node;    // DirectoryNode or TreeNode

    public MyTreeNode(string text, object node): base(text)
    {
        this.node = node;
    }
```

```
    public object Node
    {
        get
        {
            return(node);
        }
    }
}
```

The `MyTreeNode` stores the additional information in an `object` variable. For a little nicer type checking, you could add a base class for both `DirectoryNode` and `TreeNode` so that `MyTreeNode` could store a class rather than just `object`.

After this node is defined, the form's `PopulateTree()` function merely needs to have a few lines changed to create `MyTreeNode` instances rather than `TreeNode` instances.

At first glance, you may think there's an extra object for each node now, but in actuality, you've merely replaced the `TreeNode` instances with `MyTreeNode` instances, and the added overhead is merely the extra `object` field.

Populate on Demand

The next task is to change the population code so it populates only a single level. To do this, you need to remove the recursion from the `Populate()` function. This involves a bit of a subtlety; your goal is to get rid of the extra nodes below all the directories, but if you get rid of the extra nodes, there won't be any plus signs on the directories to click.

You can get around this by putting a single blank node under each directory that has files in it and setting the node of that entry to `null`. This enables the plus signs on the directory. The populate code now looks like this:

```
public void PopulateTreeNode(TreeNodeCollection treeNodeCollection,
             DirectoryNode directoryNode,
             float fractionUsed)
{
    TreeNode treeNode = new MyTreeNode(directoryNode.NameSize, directoryNode);
    treeNode.ImageIndex = FractionToIndex(fractionUsed);
    treeNode.SelectedImageIndex = treeNode.ImageIndex;
    treeNodeCollection.Add(treeNode);

    if (directoryNode.SizeTree != 0)
    {
        // Add a fake entry to this node so that there will be
        // a + sign in front of it.
        treeNode.Nodes.Add(new MyTreeNode("", null));
    }
}
```

Next, you'll need to write the function that populates the nodes below a directory when it's clicked. It looks like this:

```
public void ExpandTreeNode(MyTreeNode treeNode)
{
        // look at the first child of this tree. If the node
        // associated with it isn't null, then we've already
        // done the expansion before.
    MyTreeNode childTreeNode = (MyTreeNode) treeNode.Nodes[0];
    if (childTreeNode.Node != null)
        return;

    treeNode.Nodes.Clear();        // get rid of null entry
    DirectoryNode directoryNode = (DirectoryNode) treeNode.Node;

        // As we walk through the tree, we need to figure out the
        // percentages for each item. We do that based upon the
        // full size of this directory.
    float dirSize = directoryNode.SizeTree;
    foreach (DirectoryNode subdir in directoryNode.GetDirectories())
    {
        PopulateTreeNode(treeNode.Nodes, subdir, subdir.SizeTree / dirSize);
    }

    foreach (FileNode fileNode in directoryNode.GetFiles())
    {
        TreeNode treeFileNode = new MyTreeNode(fileNode.NameSize, fileNode);
        treeFileNode.ImageIndex = FractionToIndex(fileNode.Size / dirSize);
        treeFileNode.SelectedImageIndex = treeFileNode.ImageIndex;
        treeNode.Nodes.Add(treeFileNode);
    }
}
```

The code at the beginning of the function checks to see if the first entry in the node list has a null node. If it isn't null, then this node has been expanded before, and the tree is already up-to-date. If it is null, you delete the blank node and then add all the nodes for this directory.

Finally, you need to hook this up to the expand event. You add a handler for the `BeforeExpand` event in the Form Designer and then write the code. It's really only one line of code:

```
protected void treeView1_BeforeExpand (
            object sender, System.WinForms.TreeViewCancelEventArgs e)
{
    ExpandTreeNode((MyTreeNode) e.node);
}
```

Sorting the Files

While using the application at this point, you may notice it's tough to find the biggest files in the directory. It'd be nice to sort the files so they're listed in order from largest to smallest.

A few chapters ago, you added a `GetFiles()` function to the `DirectoryNode` class. This function returns an array of `FileNode` objects, so you can just sort the array before `GetFiles()` returns it, and that should give you the proper ordering. Nicely, the `System.Array` class has a `Sort()` member you can call to do the sorting.

For `Sort()` to work, it has to be able to figure out how to order the `FileNode` members. In the .NET Framework, you do this by implementing the `IComparable<T>` interface on `FileNode`. `IComparable<T>` has a member to compare two instances of `T` and returns integer values based on the ordering of the objects. The function for `FileNode` looks like this:

```
public int CompareTo(FileNode node2)
{
    if (this.Size < node2.Size)
        return(1);
    else if (this.Size > node2.Size)
        return(-1);
    else
        return(0);
}
```

The function compares the appropriate fields of the `FileNode` objects and returns the integer value. In this case, the `CompareTo()` function uses the `Size` property of the `FileNode` class; this simplifies the class a bit, though it does add a small bit of overhead because of the code in the `get` accessor of the property.

You then add the call to `Array.Sort()` in the `GetFiles()` function before the array is returned. That's it! It's so simple that you can add similar code to the `DirectoryNode` class so the directories are also sorted based on size.

One enhancement you could add is to allow the user to sort the files and directories either by directory or by file. See Chapter 30 for more information.

Saving and Restoring

One of the points of DiskDiff is to be able to compare the current state of a directory tree to a previous state. It's therefore important to be able to store and retrieve that state.

The logical way to do this in the .NET Framework is to use serialization. This is one of the areas in which a managed environment really shines; the runtime already has enough information in the metadata associated with a class to be able to do the serialization. You must write only the serialization code and define how you want your classes to be serialized.

The .NET Framework supports serialization either to SOAP format (the same XML format used by Web services) or to a binary format. For this example, we've decided to use SOAP formatting since it's easy to look at the resulting file and see what's happening.[3]

After adding Save and Open menu items on the File menu, add the following code in the save event handler:

3. This will become important in a few paragraphs.

```
protected void FileSave_Click (object sender, System.EventArgs e)
{
    SaveFileDialog dialog = new SaveFileDialog();
    dialog.Filter = "DiskDiff files (*.diskdiff)|*.diskdiff|" +
        "All files (*.*)|*.*";
    dialog.ShowDialog();

    Stream streamWrite = File.Create(dialog.FileName);
    SoapFormatter soapWrite = new SoapFormatter();
    soapWrite.Serialize(streamWrite, directoryNode);
    streamWrite.Close();
}
```

SaveFileDialog is a class that comes with the system, and the Filter property controls which files are shown. Once a filename is obtained from the dialog box, it's simply a matter of creating a file with that name, creating a new SoapFormatter, and calling the Serialize() function.

The open event handler is only a bit more complicated:

```
protected void FileOpen_Click (object sender, System.EventArgs e)
{
    OpenFileDialog dialog = new OpenFileDialog();
    dialog.Filter = "DiskDiff files (*.diskdiff)|*.diskdiff|"+
        "All files (*.*)|*.*";
    dialog.ShowDialog();

    try
    {
        Stream streamRead = File.OpenRead(dialog.FileName);
        SoapFormatter soapRead = new SoapFormatter();
        directoryNode = (DirectoryNode) soapRead.Deserialize(streamRead);
        streamRead.Close();
        rootDirectory = directoryNode.Root;
        treeView1.Nodes.Clear();

        PopulateTreeNode(treeView1.Nodes, directoryNode, 1.0f);
    }
    catch (Exception exception)
    {
        MessageBox.Show(exception.ToString());
    }
}
```

In this handler, the Deserialize() call reconstructs the objects in the stream passed to it. If everything goes correctly in this code, the rootDirectory field of the form is set to the top-level directory that was deserialized, and the TreeView object is populated.

Controlling Serialization

Chapter 34 covers serialization in detail. Two attributes control the behavior of the serialization:

```
[Serializable]
[NonSerialized]
```

The `Serializable` attribute is placed on classes that should be serialized, and the `NonSerialized` attribute is placed on class members that shouldn't be serialized. When doing serialization, a class is serialized only if it has the `Serialized` attribute, and each member in that class is serialized if it doesn't have the `NonSerialized` attribute.

In this example, you'll be serializing a `DirectoryNode` object, which can contain `FileNode` objects, so you'll have to annotate both of those classes. This is simple; just look through the definition of the object, and figure out which members shouldn't be saved. `DirectoryNode` looks like this:

```
[Serializable]
public class DirectoryNode: IComparable<DirectoryNode>
{
    string root;
   List<FileNode> files = new List<FileNode>();
   List<DirectoryNode> dirs = new List<DirectoryNode>();
    Directory directory;    // this directory

    long? size = null;        // size of dir in bytes
    long? sizeTree = null;        // size of dir and subdirs
    [NonSerialized]
    bool cancelled = false;
}
```

The `cancelled` field is set as `NonSerialized` since you don't need to save it.

DiskDiff uses the `BinaryFormatter` to serialize the data to disk. As well as having better performance and a smaller output size than the `SoapFormatter`, `BinaryFormatter` has been updated to support generics and nullable types. The `SoapFormatter` isn't part of the future of the .NET Framework libraries and hasn't been updated.

Finer Control of Serialization

For the `FileNode` object, the minimum amount of information to save is the full path to the file.[4] But the full path is stored in the `FileInfo` object that's part of the `FileNode` object, and you have no way to change how `FileInfo` serializes.

One option is to have the full filename stored along with the `File` object, and not serialize the file object, but that results in a duplication of data. Not only is that wasteful but you also have to keep the two variables in sync.

4. It may be possible to save just the filename and build up the full name based on the directory structure, but this would take a fair amount of code that we don't want to write right now.

The CLR serialization code provides a way for your object to take over how serialization occurs. You do this by implementing the `ISerializable` interface on your object.[5] Add the following routines to `FileNode`:

```
    // Routines to handle serialization
    // The full name and size are the only items that need
    // to be serialized.
FileNode(SerializationInfo info, StreamingContext content)
{
    file = new FileInfo(info.GetString("N"));
    size = info.GetInt64("S");
}

public void GetObjectData(SerializationInfo info, StreamingContext content)
{
    info.AddValue("N", file.FullName);
    info.AddValue("S", this.size);
}
```

In this code, the first change was to have the `FileNode` object store the size itself, rather than use the `Length` property on the `File` object.[6] The `GetObjectData()` function is implemented for the `ISerializable` interface, and it's called with each object during serialization. The function saves the values for the full name of the file and the size of the file.

To deserialize an object, the runtime creates a new instance of an object and then calls the special constructor listed previously. It extracts the two fields from the object and creates the contained `File` object.

Incidentally, the constructor isn't part of the `ISerializable` interface because constructors can't be members of interfaces. Adding this constructor is something you need to remember to do, or you'll get an exception when you try to deserialize your object.

The code changes to the `DirectoryNode` class are similar:

```
    // Routines to handle serialization
    // The full name and size are the only items that need
    // to be serialized.
DirectoryNode(SerializationInfo info, StreamingContext content)
{
    root = info.GetString("R");
    directory = new Directory(root);
    files = (List<FileNode>) info.GetValue("F", typeof(List<FileNode>));
    dirs = (List<DirectoryNode>) info.GetValue("D",
     typeof(List<DirectoryNode>));
    size = info.GetInt64("S");
    sizeTree = info.GetInt64("ST");
}
```

5. If you've used serialization in MFC, the approach used by the runtime will be quite familiar.
6. `Length` is read-only, so you have no way to set it.

```
public void GetObjectData(SerializationInfo info, StreamingContext content)
{
    info.AddValue("R", root);
    info.AddValue("F", files);
    info.AddValue("D", dirs);
    info.AddValue("S", size);
    info.AddValue("ST", sizeTree);
}
```

One possible way to further reduce the file size is to not serialize the `FileNode` objects at all but to save the names and sizes of files from within the `DirectoryNode` serialization code. This requires naming the fields `File1`, `Size1`, `File2`, `Size2`, and so on.

CHAPTER 37

Practical DiskDiff

After a long trip, you've finally arrived at the point of getting something useful done with DiskDiff. In this chapter, you'll start hooking up some useful functions and get DiskDiff ready for deployment.

Comparing Directories

Since you can now save and retrieve the state of a directory tree, it's possible to compare the saved state of a directory tree with the current state.

To do this, add code to the file-loading code so it will load the tree to a separate structure, scan the directories to get the current state of that tree, and then perform a comparison between the two trees. The code in the open handler is as follows:

```
    // deserialize the save version, and read in the new version
this.statusBar1.Text = "Opening...";
directoryNodeBaseline = DirectoryNode.Deserialize(dialog.FileName);
directoryNode = directoryNodeBaseline;
rootDirectory = directoryNodeBaseline.Root;
DoTree();
```

The `directoryNodeBaseline` variable holds the baseline version, and `DoTree()` is called to get the current state of that directory. In this code, we then modified the `DoTreeDone()` function to call a comparison function.

You then add the comparison function `CompareTrees()` to the `DirectoryNode` class. This function matches file and directory entries in the current and baseline trees. For each match, it computes the ratio of the space used in the trees and stores the ratio as part of the object. The ratio is then displayed as part of the file and directory entries, and the icon displayed for an entry is red if the entry is bigger than the baseline entry.

The DiskDiff application now looks like Figure 37-1.

This baseline view is of the project directory. We then added the `a.a` file to the directory and loaded the baseline, creating this view. Note that `a.a` is flagged as a new file and is presented using an icon with a red background. The root level shows the increase in size of the entire tree and also appears with a red background.

You can now use DiskDiff for its intended use: to identify which directories have increased usage. In the current version, however, you have no way to perform the file operations to clean up the issues.

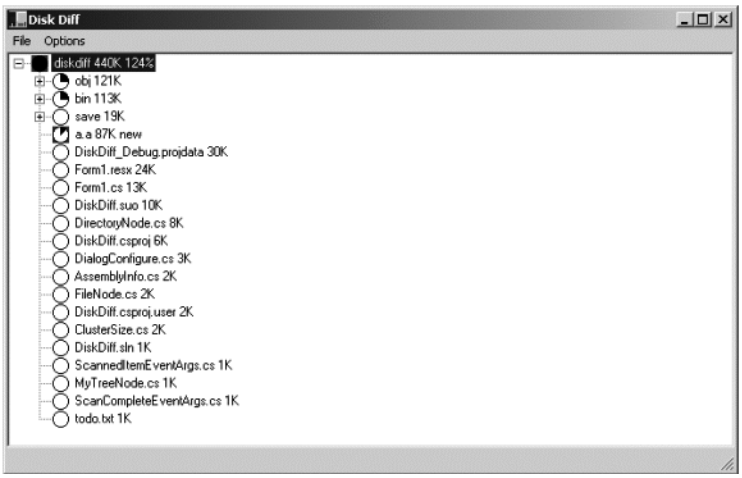

Figure 37-1. *Directory tree with differences flagged*

File Manipulation

To be able to clean up the extra junk on a drive, you need to be able to delete files from your interface. The usual user interface to use in this situation is a context menu.

Adding a context menu to a `TreeView` is simple. Drag a context menu control from the Toolbox to the form. Then, select the `TreeView` object, go to the property window, and set the `ContextMenu` property to the context menu you just added.

Next, you'll need to set up the contents of the menu. You can do this in the form constructor after the call to `InitializeComponent()`. In this case, you add a helper function to add items to the menu, since all the menu items have the same handler:

```
void AddContextMenuItem(string itemString)
{
    MenuItem menuItem;
    menuItem = new MenuItem(itemString,
                new EventHandler(treeContextMenuClick));
    treeContextMenu.MenuItems.Add(menuItem);
}
```

Next, add the following lines to the constructor for the form:

```
AddContextMenuItem("Filename");
AddContextMenuItem("Delete");
AddContextMenuItem("Delete Contents");
AddContextMenuItem("-");
AddContextMenuItem("View in Notepad");
AddContextMenuItem("Launch");
```

The fourth line gives you a divider in the menu. You should also define some constants so you know what the indices of the menu items are:

```
const int menuIndexFilename = 0;
const int menuIndexDelete = 1;
const int menuIndexDeleteContents = 2;
const int menuIndexViewInNotepad = 4;
const int menuIndexLaunch = 5;
```

That gets the initial plumbing hooked up. Now it's time to write some code to customize the menu before it opens. In this case, you want to add the name of the file or directory as the first entry of the menu and enable/disable the menu items based on the item that was selected. Here's the code we wrote initially:

```
protected void treeContextMenu_Popup (object sender, System.EventArgs e)
{
    MyTreeNode treeNode = (MyTreeNode) treeView1.SelectedNode;

    string filename = null;
    bool deleteContents = false;
    bool viewInNotepad = false;
    bool launch = false;
    if (treeNode.Node is DirectoryNode)
    {
        DirectoryNode directoryNode = (DirectoryNode) treeNode.Node;
        filename = directoryNode.Name;
        deleteContents = true;
        viewInNotepad = false;
    }

    if (treeNode.Node is FileNode)
    {
        FileNode fileNode = (FileNode) treeNode.Node;
        filename = fileNode.Name;
        deleteContents = false;
        viewInNotepad = true;
    }
    treeContextMenu.MenuItems[menuIndexFilename].Text = filename;
    treeContextMenu.MenuItems[menuIndexDeleteContents].Enabled =
                    deleteContents;
    treeContextMenu.MenuItems[menuIndexViewInNotepad].Enabled =
                    viewInNotepad;
}
```

This code works, but the two `if` blocks are a good example of a poor design choice. The code for `DirectoryNode` and `FileNode` is essentially identical, but you still have to write separate code for each type of entry.

Now is a good time to define an abstract class for the properties that are shared. It will be called `BaseNode` and will have abstract properties for the properties that `DirectoryNode` and `FileNode` share. A few properties will have the same implementation in `BaseNode` and `DirectoryNode`, which is the reason an abstract class is used instead of an interface. It turns out there are quite a few of them, which is another good indication that this is a good change to make:

```
public abstract class BaseNode
{
    public abstract long Size { get; }
    public abstract string Name  { get; }
    public abstract string FullName {get; }
    public abstract string NameSize { get; }
    public abstract bool FlagRed { get; }
    public abstract bool EnableDeleteContents { get; }
    public abstract bool EnableViewInNotepad { get; }
}
```

Modify the `FileNode` and `DirectoryNode` classes to override these properties. This reduces the code in the handler from 18 lines to only 4:

```
protected void treeContextMenu_Popup (object sender, System.EventArgs e)
{
    BaseNode baseNode = ((MyTreeNode)treeView1.SelectedNode).Node;

    treeContextMenu.MenuItems[menuIndexFilename].Text =
        baseNode.Name;
    treeContextMenu.MenuItems[menuIndexDeleteContents].Enabled =
        baseNode.EnableDeleteContents;
    treeContextMenu.MenuItems[menuIndexViewInNotepad].Enabled =
        baseNode.EnableViewInNotepad;
}
```

File and Directory Operations

Now it's time to actually implement the file and directory operation. You do this in the event handler for the context menu's items. Here's the initial version:

```
protected void treeContextMenuClick(object sender, EventArgs e)
{
    MenuItem menuItem = (MenuItem) sender;
    BaseNode baseNode = ((MyTreeNode) treeView1.SelectedNode).Node;

    switch (menuItem.Index)
    {
        case menuIndexDelete:
            baseNode.Delete();
            break;
        case menuIndexDeleteContents:
            ((DirectoryNode)baseNode).DeleteContents();
            break;
```

```
            case menuIndexViewInNotepad:
                ProcessStartInfo processStartInfo = new ProcessStartInfo();
                processStartInfo.FileName = "notepad";
                processStartInfo.Arguments = baseNode.FullName;
                Process.Start(processStartInfo);
                break;
            case menuIndexLaunch:
                try
                {
                    processStartInfo = new ProcessStartInfo();
                    processStartInfo.FileName = baseNode.FullName;
                    Process.Start(processStartInfo);
                }
                catch (Exception exc)
                {
                    MessageBox.Show(exc.ToString());
                }
                break;
        }
    }
```

We'll cover each of these menu items in a little more detail in the following sections:

- **Delete**: The Delete item calls a virtual function in the `FileNode` and `DirectoryNode` classes to delete the file or directory. This virtual function will call either `File.Delete()` or `Directory.Delete()`.
- **Delete Contents**: The Delete Contents item is done by a function in the `DirectoryNode` class.
- **View in Notepad and Launch**: These two items are handled by starting a separate process. The process is started by using the `Process` class in the `System.Diagnostics` namespace, as discussed in Chapter 34.

Updating the User Interface

When either of the delete options is used, each works correctly, but the `TreeView` object doesn't correctly reflect the updated structure and neither does the internal structure.

We'll attack these separately, handling the internal structure first (for reasons that will soon become apparent). To delete a file or directory from the internal structure, you need to find the parent of that item and then delete the node from the parent's file or directory list (as appropriate).

The internal structure doesn't have any back reference, so if you were to do the delete based only on that structure, you'd have to search for the node from the root. Luckily, the `TreeNode` objects do have such links, so you'll merely get the parent of the selected node and then tell the parent `DirectoryNode` to delete the selected node.

Deleting the contents of a directory is easier; you merely modify the `DeleteContents()` method on `DirectoryNode` to clear the `files` and `dirs` arrays.

That finishes updating the nodes of the internal structure, but now that the nodes have been removed, the cached sizes along the modified path (from the deleted nodes up to the root) are incorrect. To update these, we originally planned on traversing the modified path and modifying the sizes based on the changed values. After a bit of reflection, however, we decided to do the brute-force method and added the `ClearSizeCache()` to the `DirectoryNode` class; this method merely resets the sizes on all the directories to null, forcing them to be recalculated when they're requested through the properties.

You're now ready to update the tree so it reflects the new state. Removing a node from the `TreeView` is as simple as calling the `Remove()` function.

Finally, you need to update the sizes on the displayed nodes. You can do this in the `UpdateTreeNodes()` member on the form. This member takes a collection of nodes and updates the text for each of them (which will indirectly cause the sizes to be recalculated).

That's it—you now can delete files from the structure and have the `TreeView` always show you the proper information.

However, if you load an existing baseline and then delete files, you'll notice that the comparison is no longer correct.

A Bit of Refactoring

Our initial view of the display of ratios (and the color-coding) of directories and files was that they'd be static, which was why the `DirectoryNode` stored the ratio rather than calculating it when needed. In retrospect, this was a poor decision, since we knew we'd want to delete files later. *Mea culpa*.

After exploring updating the ratio when necessary (an ugly task), we decided to have the `DirectoryNode` store a reference to the corresponding baseline `DirectoryNode` and then calculate the ratio when it was requested.[1]

The change to the `DirectoryNode` class simplifies the code elsewhere, and the percentages are now updated correctly. Another problem appears, however; the "pie chart" coding of the icons doesn't get updated correctly when a file is deleted. A look at the current code shows that the form calculates the fraction when the tree is populated. Adding code to update this—and to get it right—seems to be decidedly nontrivial.

An alternate approach is to have a file or directory calculate the fraction directly, based on the parent's size. Doing this will require that files and directories be able to easily get to their parents, and the easiest way to do that is to add a `Parent` property to the `BaseNode` class. This enables calculating the fraction on the fly and finally allows you to get the behavior you want. It also allows a directory or file to be able to delete itself from the parent directory, which eliminates the code that walked the tree to do this.

1. This works because you have no way to add files in this interface, which requires rewalking the baseline tree to check for a corresponding baseline file (or directory).

Cleaning Up for the Parents

Although you may not mind if your friends see a big mess, you'd probably prefer that things are nicer for your parents. In the following sections, you'll add a few things to make your application nicer.

Keyboard Accelerators

Accelerators are quite easy to add to an application; you just put an ampersand in front of whatever character is the accelerator key in the text.

Most Recently Used List

The most recently used (MRU) list is a nice way to access recently used documents. On the File menu, you can add entries (typically four) to hold the last four documents used and then use these to open those documents.

You can do the user interface side of this as you'd do any other menu item, but to store the items themselves, you'll need to use the Windows Registry. For this example, you'll encapsulate this access in the following class:

```
public class MRU
{
    ArrayList entries = new ArrayList();
    string keyRoot = @"Software\Sample\DiskDiff";

    public MRU()
    {
        RegistryKey ourKey;
        ourKey = Registry.CurrentUser.CreateSubKey(keyRoot);

        for (int index = 0; index < 4; index++)
        {
            string keyName = "MRU_" + index;
            string value = (string) ourKey.GetValue(keyName);
            if (value != null)
            {
                entries.Insert(index, value);
                Console.WriteLine("{0} {1}", index, value);
            }
        }
        ourKey.Close();
    }
```

```
    public string this[int index]
    {
        get
        {
            if (index >= entries.Count)
                return("");
            else
                return((string) entries[index]);
        }
    }

    public void AddEntry(string entry)
    {
        entries.Insert(0, entry);
        if (entries.Count > 4)
        {
            entries.RemoveAt(4);
        }

        Save();
    }

    void Save()
    {
        RegistryKey ourKey;
        ourKey = Registry.CurrentUser.CreateSubKey(keyRoot);

        for (int index = 0; index < entries.Count; index++)
        {
            string keyName = "MRU_" + index;
            ourKey.SetValue(keyName, entries[index]);
        }
        ourKey.Close();
    }
}
```

You can think of the Windows Registry as a hierarchical database in which a program can store values. We're storing our information in the `Software\Sample\DiskDiff` key off the `HKEY_CURRENT_USER` root (which stores per-user customization).[2]

To access information in the Windows Registry, you must first open a key at the specific level. In the constructor, the key is open and then each of the keys is looked up. To save the data to the Windows Registry, the process is reduced. The indexer retrieves the current values of the list, and the `AddEntry()` function adds an entry to the first entry of the list.

2. The typical pattern is `Software/<Company>/<Program>`, but we don't have a company to use here.

You then hook up this class into the rest of the application. The code to save an item now adds an entry to the MRU list, and you add a menu item handler to open the file when one of the items is chosen.

Using the Windows Registry is the traditional way of storing such information, but the .NET Framework provides a new method, using a config file, which is covered in the next section. The reason we present both alternatives in this chapter is that the Windows Registry is still a valid implementation option despite being mostly superceded by configuration files. The Windows Registry is more mature as a user setting store and generally works better in scenarios with roaming users. You can configure a Windows security domain so the `HKEY_CURRENT_USER` section of the Windows Registry is automatically downloaded to any machine that a user logs into; this way, their configuration settings are always available. By contrast, configuration files don't have this level of support yet.

Most Recently Used List: A Configuration File Alternative

New to .NET 2.0 is the ability to use the configuration infrastructure to store per-user settings. Achieving this is simple, and it offers an alternative to the Windows Registry if you don't require support for roaming users. To begin setting up a configuration file for DiskDiff, open the Properties page for the project, select the Settings tab, and add an MRU setting of type `string` with a User scope, as shown in Figure 37-2.

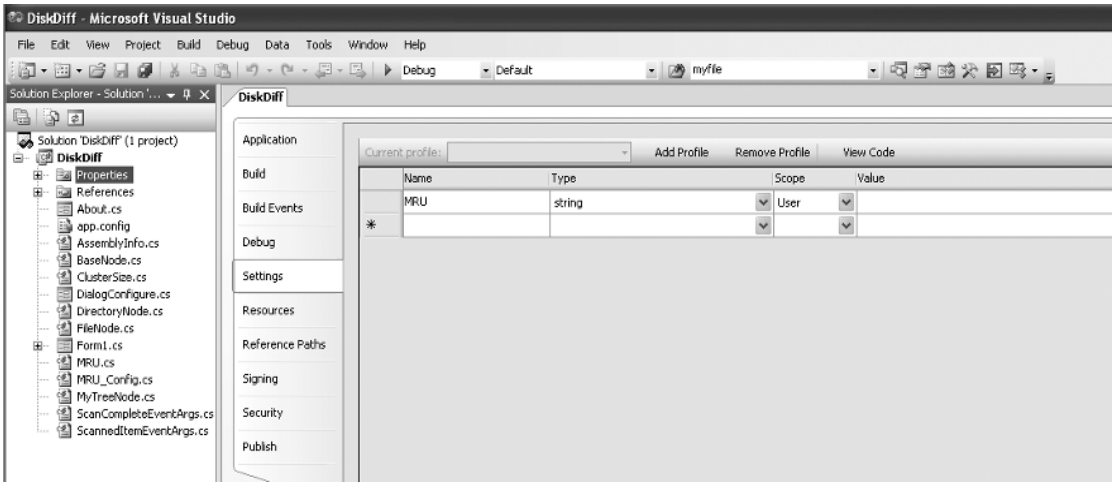

Figure 37-2. *MRU setting of type `string` with a User scope*

You can now write the configuration file–based MRU class using the same interface as the registry-based approach. In this case, the filenames are stored in a single setting and delineated with pipes:

```
using System;
using System.Collections.Generic;
using System.Configuration;
using DiskDiff.Properties;

namespace DiskDiff
{
 class MRU_Config
 {
  List<String> entries = new List<string>();
  Settings settings = new Settings();

  public MRU_Config()
  {
   entries.AddRange(settings.MRU.Split('|'));
  }

  public string this[int index]
  {
   get
   {
    if (index >= entries.Count)
     return ("");
    else
     return entries[index];
   }
  }

  public void AddEntry(string entry)
  {
   entries.Insert(0, entry);
   if (entries.Count > 4)
   {
    entries.RemoveAt(4);
   }

   Save();
  }
```

```
  void Save()
  {
   foreach(string s in entries){
    settings.MRU += (s + "|");
   }
   settings.Save();
  }
 }
}
```

To modify DiskDiff to use the configuration file approach, you need to change a single line of code in Form1:

```
//was MRU mru = new MRU();
MRU_Config mru = new MRU_Config();
```

ToolTips

ToolTips make your user interface much easier to figure out and reduces the need for help. Adding them is easy in a Windows Forms application. Dragging a `ToolTip` control from the Toolbox to the form adds a `ToolTip` property to the properties collection of each control on the form. Merely set the `ToolTip` control's text for each property, and you're done.

Increased Accuracy

The code still has one problem. Your goal is to keep track of disk space usage, but the program is tracking the size of the files, not the disk space used by those files. The difference has to do with the way file systems work.

Each disk on your computer has what's known as the *cluster size*, which is the unit of allocation for the files on that disk. When space for a file is allocated, a sufficient number of clusters is allocated to hold the contents of the file. This means if you have a disk with a cluster size of 4,096 bytes, a file with a single byte in it still occupies a full 4,096 bytes of space.

For the purposes of this application, the effect of this can be considerable. If you have a file that has a thousand 300-byte files, the current implementation will total 300,000 bytes. Depending on the cluster size, however, the usage could be quite a bit more. The current cluster size on our system is 512 bytes, so the actual usage is 512,000 bytes. This cluster size is pretty small; a typical cluster size on an NTFS system is 4KB, which means you'd be using 4MB of space to store 300KB bytes.

It can get worse than this if you're running the FAT16 file system. A FAT16 system can have only 64KB clusters on a disk, so if your disk size is 2GB, your cluster size is 32KB. In the previous example, this means you'd be using 32MB of space to store 300KB of file.

It's therefore useful to factor the cluster size into determining the actual amount of space used by the files in a directory. The first thing to do is to figure out how to get the cluster size for a disk.

You can access the cluster size for a disk by using the `GetDiskFreeSpace()` function. This is a Win32 function, so you'll need to use platform invoke to call it. It's nicest if you encapsulate this function in a class, which is as follows:

```
public class ClusterSize
{
    private ClusterSize() {}

    public static int GetClusterSize(string root)
    {
        int sectorsPerCluster = 0;
        int bytesPerSector = 0;
        int numberOfFreeClusters = 0;
        int totalNumberOfClusters = 0;
        Console.WriteLine("GetFreeSpace: {0}", root);
        bool result = GetDiskFreeSpace(
            root,
            ref sectorsPerCluster,
            ref bytesPerSector,
            ref numberOfFreeClusters,
            ref totalNumberOfClusters);
                    return(sectorsPerCluster * bytesPerSector);
    }

    [DllImport("kernel32.dll", SetLastError=true)]
    static extern bool GetDiskFreeSpace(
        string rootPathName,
        ref int sectorsPerCluster,
        ref int bytesPerSector,
        ref int numberOfFreeClusters,
        ref int totalNumberOfClusters);
}
```

The declaration for the function is at the end of the class, and it follows the usual `PInvoke` format. The function gets the cluster size of a disk and returns the value.

This function works, but it has a few problems. The first one is that it'd be nicer to pass in a full directory rather than a disk name. The second one is that it'd be convenient for every directory to call and get this function, but you don't want to call the function every time. In other words, you need to cache the value. You do this by keeping a static hash table that stores the cluster sizes for disks and then checking it calling the function. Add the following lines to the `GetClusterSize()` function:

```
string diskName = root.Substring(0, 1) + @":\";
object lookup;
lookup = sizeCache[diskName];

if (lookup != null)
    return((int) lookup);
```

Switching to Use Cluster Size

Now that you have a way to get the cluster size for a disk, you can modify the main program to use this function. The code will support both the allocated and used sizes so you'll have the option of (somehow) displaying both.

The first change is to the `FileNode` class. It will now store both sizes and determine their values in the constructor:

```
this.sizeUsed = file.Length;
long clusterSize = ClusterSize.GetClusterSize(file.FullName);
this.size = ((sizeUsed + clusterSize - 1) / clusterSize) * clusterSize;
```

A bit of explanation is probably in order. To figure out the allocated size of this file, you need to round the size to the next multiple of the cluster size. The first step is to determine the number of clusters, which you can do by adding one less than the cluster size to the size and then dividing it (an integer division) to get the number of clusters.

Whether this works is easy to determine by considering the boundary conditions. Assuming a cluster size of 512, a file that's 1 byte long will occupy 512 bytes:

```
((1 + 511) / 512) * 512
```

Similarly, a file that's 512 bytes will occupy 512 bytes:

```
((512 + 511) / 512) * 512
```

and a file of size 513 bytes will occupy 1024 bytes:

```
((513 + 511) / 512) * 512
```

Now that you've updated the `FileNode` object, you can also update the `DirectoryNode` class. You can add a `SizeUsed` property and add the `UpdateTreeSizes()` member to update both values when necessary. You can also take this opportunity to remove some of the code that tried to calculate these values during the file scan; it turns out to be more of a hassle than it was worth maintaining the code in both places.

Deploying DiskDiff

Windows Forms applications have two major disadvantages compared to technologies such as ASP.NET and Web Forms that can deliver user interfaces via the browser: cross-platform support and deployment issues. .NET goes some way to addressing these "Web vs. Windows" problems; third-party .NET implementations such as Mono (`http://www.go-mono.com/`) and

DotGNU (`http://www.dotgnu.org/`) allow Windows Forms applications to be run on a wide variety of platforms, and the deployment model of .NET improves with each new version of the .NET Framework. This section won't cover deployment in detail but rather will provide a short glimpse of the ease of Windows Forms deployment offered by a new technology in .NET 2.0 called ClickOnce.

Before covering ClickOnce, though, it's worth briefly discussing desktop application deployment through the ages. The first Windows applications didn't require much in the way of installation—they could be simply copied to some location on the hard drive or run from a floppy disk. To provide a simpler experience for newer users, installation programs that added shortcuts that allowed the application to be launched without navigating to the EXE file became popular. With Windows 95, the registry replaced INI files as the preferred location to store configuration data, and installers generally added a number of registry settings. As more features were added to Windows, installers became more complex in an attempt to deal with these features; dealing with COM registration, MDAC versioning, event log sources, and COM+ applications all became part of the installation activities.

.NET took a back-to-the-future approach with Windows Forms applications. XCOPY deployment, where the EXE and DLL files were simply copied onto disk and run, was billed as a major improvement. XCOPY fell short in many areas—who created the shortcuts in the Start menu, where do per-user settings go, and how does a new version of the application get installed?

An alternative to XCOPY was trickle-down deployment, where the use of the `Assembly.LoadFrom()` method bought assemblies down across the network or Internet. The trickle-down application had a bootstrap application that was installed via XCOPY or a traditional installer, and then the real application was pulled down on demand in a similar way to the Web deployment model. Trickle-down deployment was a good start but had a number of shortfalls. Code Access Security (CAS) was a problem—if the application did things such as use interop, the user's machine needed to be permanently connected to the server that hosted the trickle-down assemblies. Retrofitting trickle-down to an application that was implemented without it was a huge undertaking.

Various third-party and product team solutions offered various aspects of the traditional install, XCOPY, and trickle-down models. The Updater Application Block produced by the Windows Forms team and Patterns and Practices group at Microsoft was one of the more successful attempts, and this has formed the foundation of ClickOnce.

ClickOnce is a deployment technique that has direct .NET Framework and Visual Studio support. You can deploy a project with ClickOnce by right-clicking the project node in the Visual Studio Solution Explorer and selecting Publish.[3] The first page in the wizard allows you to select the publication location, as shown in Figure 37-3.

The next step is to determine the mode of application usage. You can install an application in online mode, which means it can be run only from the location it was published to and no Start menu shortcuts will be installed. Online-only mode is a lightweight model and is best for an application that a user will need to start only a couple of times. The other application mode is online/offline. In this mode, the install is more like a traditional application install, with

3. You can use .NET Framework SDK tools such as the Manifest Generation and Editing Tool (`Mage.exe`) to build a ClickOnce deployment, but we won't cover them here. See the MSDN Library documentation for more details.

Start menu and Add/Remove Program entries added. Because DiskDiff fits into the mode of a more traditional application, the online/offline usage model makes the most sense.

Figure 37-3. *ClickOnce publish location*

Once you've completed the Publish Wizard, you can deploy the application to the nominated publication location. End users will typically be notified of the application's location by e-mail or a Web page, and then they can navigate to the application via a URL. If the application is an online-only application, and the permissions needed to run the application are granted to the zone that the application is installed from, the application will simply execute when the user clicks the hyperlink. If the prerequisites required to run the application (such as the .NET Framework) aren't installed, the user will be prompted to install these first.

For online/offline applications, an installation wizard will guide the user through the installation process. For DiskDiff, using interop exceeds the permission set that would be granted to an application that originates from an intranet or Internet, and the end user is asked if they want to increase the permissions granted to the application. Figure 37-4 shows the warnings you'll see for a DiskDiff installation.

One of the key benefits of deploying via ClickOnce is that you can update applications without having to check for or download updates. Online-only applications will always be up-to-date, as the end user is effectively running the application directly off the remote server. (However, a local cached copy will be maintained in much the same way that Web pages can be cached.) For online/offline applications, an application settings file called a *manifest*, which is produced when an application is published, will control when and how often checks are made for newer versions of the application.

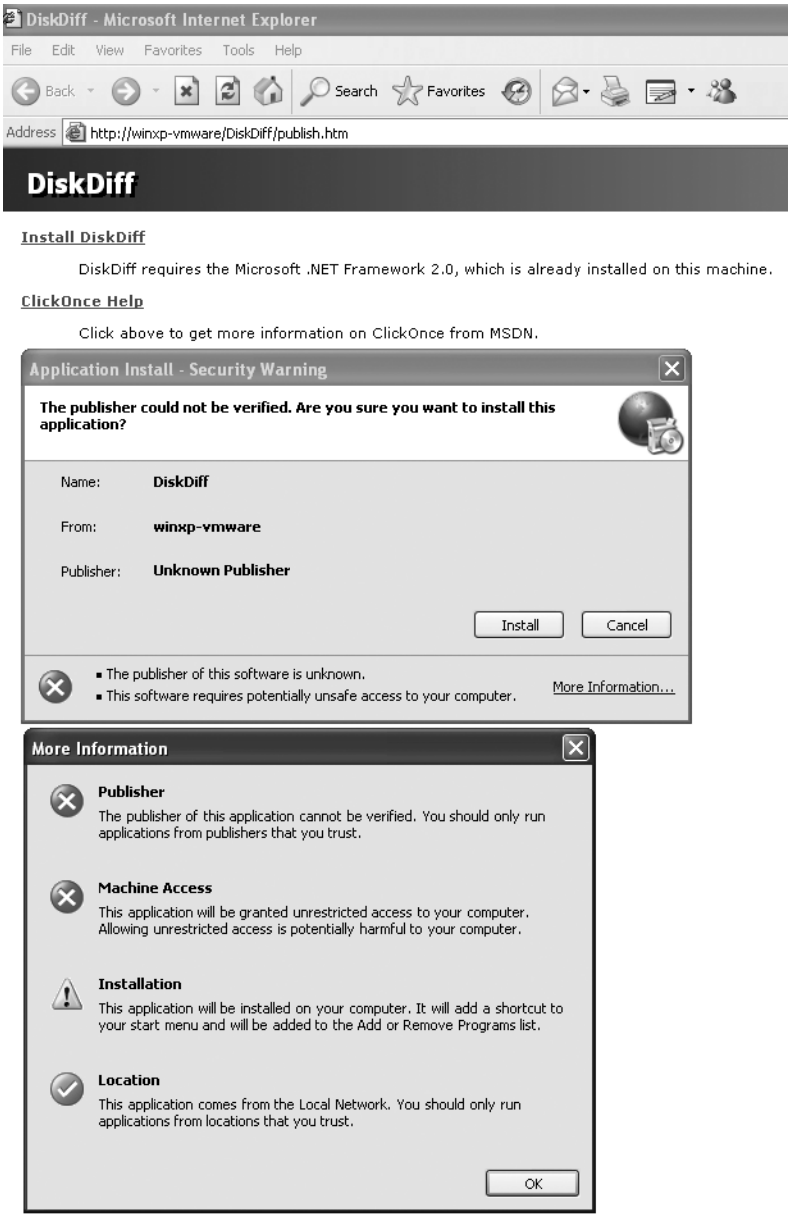

Figure 37-4. *DiskDiff installation*

ClickOnce in Perspective

ClickOnce solves many of the problems traditionally associated with desktop applications. By combining the ease and updatability of Web deployment with the richness of a traditional application, Windows Forms aims to provide the optimum client tier for a wide range of scenarios.

ClickOnce doesn't make other deployment techniques such as XCOPY and MSI-based deployments redundant. XCOPY is still the simplest deployment options, and ClickOnce doesn't offer the wide range of functionality that traditional installer technology offers. If a deployment needs to make significant changes to a system, such as installing a device driver or updating an operating system feature, you'll still need a traditional installer. Windows is moving toward the managed code model in its Longhorn release, so this will reduce the need for MSI-based technologies.

CHAPTER 38

Deeper into C#

This chapter delves deeper into some issues you might encounter using C#. It covers some topics of interest to the library/framework author, such as style guidelines and XML documentation, and it also discusses how to write unsafe code and how the .NET runtime's garbage collector works.

C# Style

Most languages develop an expected idiom for expression. When dealing with C character strings, for example, the usual idiom involves pointer arithmetic rather than array references. C# hasn't been around long enough for programmers to have lots of experience in this area, but the .NET CLR has some guidelines you should consider.

The "Class Library Design Guidelines" section in the .NET documentation details these guidelines, which are especially important for framework or library authors.

The examples in this book conform to the guidelines, so they should be fairly familiar already. The .NET CLR classes and samples also have many examples.

Naming

You can use two naming conventions:

- PascalCasing capitalizes the first character of each word.
- camelCasing is the same as PascalCasing, except the first character of the first word isn't capitalized.

In general, use PascalCasing for anything that's visible externally from a class, such as classes, enums, methods, and so on. The exception to this is method parameters, which you define using camelCasing.

You define private members of classes, such as fields, using camelCasing.

A few other conventions exist in naming:

- Avoid common keywords in naming to decrease the chance of collisions in other languages.
- End event classes with `EventArgs`.
- End exception classes with `Exception`.

- Start interfaces with `I`.
- End attribute classes in `Attribute`.

Hungarian naming (prefixing the name of the variable with the type of the variable) is discouraged for C# code because the added information about the variable isn't as important as making the code easier to read. For example, `strEmployeeName` is tougher to read than `employeeName`, and pointer use is rare in C#.

Conventions such as adding `m_` or `_` at the beginning of fields to denote that the field belongs to an instance is a matter of personal taste, though the usual convention in sample code is to use simple names without prefixes for fields.

Encapsulation

In general, you should heavily encapsulate classes. In other words, a class should expose as little of its internal architecture as possible.

In practice, this means using properties rather than fields to allow for future changes.

Guidelines for Library Authors

The following guidelines are useful to programmers who are writing libraries that will be used by others.

CLS Compliance

When writing software that will be consumed by other developers, it makes sense to comply with the CLS. This specification details what features a language should support to be a .NET-compliant language; you can find it in the "What Is the Common Language Specification" section of the .NET SDK documentation.

The C# compiler will check code for compliance if the `ClsCompliant` assembly attribute is placed in one of the source files.

To be CLS compliant, you have the following restrictions:

- Unsigned types can't be exposed as part of the public interface of a class. You can use them freely in the private part of a class.
- Unsafe (for example, pointer) types can't be exposed in the public interface of the class. As with unsigned types, you can use them in the private parts of the class.
- Identifiers (such as class names or member names) can't differ only in case.

For example, compiling the following will produce an error:

```
// error
using System;

[CLSCompliant(true)]
```

```
class Test
{
    public uint Process() {return(0);}
}
```

Class Naming

To help prevent collisions between namespaces and classes provided by different companies, you should name namespaces using the `CompanyName.TechnologyName` convention. For example, the full name of a class to control an X-ray laser would be something like this:

```
AppliedEnergy.XRayLaser.Controller
```

Unsafe Context

Code verification has many benefits in the .NET runtime. Being able to verify that code is type-safe not only enables download scenarios but it also prevents many common programming errors.

When dealing with binary structures or talking to COM objects that take structures containing pointers, or when performance is critical, you'll need more control. In these situations, you can use unsafe code.

Unsafe means that the runtime can't verify the code is safe to execute. It therefore can be executed only if the assembly has full trust, which means it can't be used in download scenarios, preventing abuse of unsafe code for malicious purposes.

The following is an example of using unsafe code to copy arrays of structures quickly. The structure being copied is a point structure consisting of x and y values.

Three versions of the function that clones arrays of points exist. `ClonePointArray()` is written without using unsafe features and merely copies the array entries. The second version, `ClonePointArrayUnsafe()`, uses pointers to iterate through the memory and copy it. The final version, `ClonePointArrayMemcpy()`, calls the system function `CopyMemory()` to perform the copy.

To give some time comparisons, use the following code:

```
// file=unsafe.cs
// compile with: csc /unsafe /o+ unsafe.cs
using System;
using System.Diagnostics;
using System.Runtime.InteropServices;
class Counter
{
    public static long Frequency
    {
        get
        {
            long freq = 0;
            QueryPerformanceFrequency(ref freq);
            return freq;
        }
    }
```

```
    public static long Value
    {
        get
        {
            long count = 0;
            QueryPerformanceCounter(ref count);
            return count;
        }
    }

    [System.Runtime.InteropServices.DllImport("KERNEL32",
    CharSet=System.Runtime.InteropServices.CharSet.Auto)]
    private static extern bool QueryPerformanceCounter(
        ref long lpPerformanceCount);

    [System.Runtime.InteropServices.DllImport("KERNEL32",
    CharSet=System.Runtime.InteropServices.CharSet.Auto)]
    private static extern bool QueryPerformanceFrequency( ref long lpFrequency);
}

public struct Point
{
    public Point(int x, int y)
    {
        this.x = x;
        this.y = y;
    }

        // safe version
    public static Point[] ClonePointArray(Point[] a)
    {
        Point[] ret = new Point[a.Length];

        for (int index = 0; index < a.Length; index++)
            ret[index] = a[index];

        return(ret);
    }

        // unsafe version using pointer arithmetic
    unsafe public static Point[] ClonePointArrayUnsafe(Point[] a)
    {
        Point[] ret = new Point[a.Length];

            // a and ret are pinned; they cannot be moved by
            // the garbage collector inside the fixed block.
```

```
        fixed (Point* src = a, dest = ret)
        {
            Point*    pSrc = src;
            Point*    pDest = dest;
            for (int index = 0; index < a.Length; index++)
            {
                *pDest = *pSrc;
                pSrc++;
                pDest++;
            }
        }

        return(ret);
    }
        // import CopyMemory from kernel32
    [DllImport("kernel32.dll")]
    unsafe public static extern void
    CopyMemory(void* dest, void* src, int length);

    // unsafe version calling CopyMemory()
    unsafe public static Point[] ClonePointArrayMemcpy(Point[] a)
    {
        Point[] ret = new Point[a.Length];

        fixed (Point* src = a, dest = ret)
        {
            CopyMemory(dest, src, a.Length * sizeof(Point));
        }

        return(ret);
    }

    public override string ToString()
    {
        return(String.Format("({0}, {1})", x, y));
    }

    int x;
    int y;
}

class Test
{
    const int iterations = 20000;    // # to do copy
    const int points = 1000;         // # of points in array
    const int retryCount = 5;        // # of times to retry
```

```
    public delegate Point[] CloneFunction(Point[] a);

    public static void TimeFunction(Point[] arr,
        CloneFunction func, string label)
    {
        Point[]    arrCopy = null;
        long start;
        long delta;
        double min = 5000.0d;    // big number;

            // do the whole copy retryCount times, find fastest time
        for (int retry = 0; retry < retryCount; retry++)
        {
            start = Counter.Value;
            for (int iterate = 0; iterate < iterations; iterate++)
                arrCopy = func(arr);
            delta = Counter.Value - start;
            double result = (double) delta / Counter.Frequency;
            if (result < min)
                min = result;
        }
        Console.WriteLine("{0}: {1:F3} seconds", label, min);
    }

    public static void Main()
    {
        Console.WriteLine("Points, Iterations: {0} {1}", points, iterations);
        Point[] arr = new Point[points];
        for (int index = 0; index < points; index++)
            arr[index] = new Point(3, 5);

        TimeFunction(arr,
            new CloneFunction(Point.ClonePointArrayMemcpy), "Memcpy");
        TimeFunction(arr,
            new CloneFunction(Point.ClonePointArrayUnsafe), "Unsafe");
        TimeFunction(arr,
            new CloneFunction(Point.ClonePointArray), "Baseline");
    }
}
```

The timer function uses a delegate to describe the clone function, so it can use any of the clone functions. It uses a `Counter` class, which provides access to the system timers. The accuracy of this class will vary based upon the version of Windows that's being used.

As with any benchmarking, the initial state of memory is important. To help control this, `TimeFunction()` executes each method five times and prints only the shortest time. Typically, the first iteration is slower; the CPU cache isn't ready yet, so subsequent times get faster. The times listed in Table 38-1 were generated on a 600MHz Pentium III laptop running Windows 2000

Professional, but they were generated with beta software, so the performance probably isn't indicative of the performance of the final product.

The program was run with several different values for `points` and `iterations`.

Table 38-1. *Benchmark Results for Different Array Cloning Techniques*

Method	p=10, i=2,000,000	p=1,000, i=20,000	p=100,000, i=200
Baseline	0.775	0.506	2.266
Unsafe	0.754	0.431	2.266
Memcpy	1.101	0.315	2.121

For small arrays, the unsafe code is the fastest, and for large arrays, the system call is the fastest. The system call loses on smaller arrays because of the overhead of calling into the native function. The interesting part is that the unsafe code isn't a clear win over the baseline code.

The lessons in all this are that unsafe code doesn't automatically mean faster code and it's important to benchmark when doing performance work.

XML Documentation

Keeping documentation synchronized with the actual implementation is always a challenge. One way of keeping it up-to-date is to write the documentation as part of the source and then extract it into a separate file.

C# supports an XML-based documentation format. It can verify that the XML is well-formed, do some context-based validation, add some information that only a compiler can get consistently correct, and write it out to a separate file.

You can divide C# XML support into two sections: compiler support and documentation convention. In the compiler support section, the tags are specially processed by the compiler for verification of contents or symbol lookup. The remaining tags define the .NET documentation convention and are passed through unchanged by the compiler.

Compiler Support Tags

The compiler-support tags are a good example of compiler magic; they're processed using information that's known only to the compiler. The following example illustrates how to use the support tags:

```
// file: employee.cs
using System;
namespace Payroll
{

/// <summary>
/// The Employee class holds data about an employee.
/// This class contains a <see cref="String">string</see>
/// </summary>
```

```
public class Employee
{
    /// <summary>
    /// Constructor for an Employee instance. Note that
    /// <paramref name="name">name2</paramref> is a string.
    /// </summary>
    /// <param name="id">Employee id number</param>
    /// <param name="name">Employee Name</param>
    public Employee(int id, string name)
    {
        this.id = id;
        this.name = name;
    }

    /// <summary>
    /// Parameterless constructor for an employee instance
    /// </summary>
    /// <remarks>
    /// <seealso cref="Employee(int, string)">Employee(int, string)</seealso>
    /// </remarks>
    public Employee()
    {
        id = -1;
        name = null;
    }
    int id;
    string name;
}
}
```

The compiler performs special processing on four of the documentation tags. For the `param` and `paramref` tags, it validates that the name referred to inside the tag is the name of a parameter to the function.

For the `see` and `seealso` tags, it takes the name passed in the `cref` attribute and looks it up using the identifier lookup rules so the name can be resolved to a fully qualified name. It then places a code at the front of the name to tell what the name refers to. For example, the following:

```
<see cref="String">
```

becomes this:

```
<see cref="T:System.String">
```

`String` resolved to the `System.String` class, and `T:` means that it's a type.

The `seealso` tag is handled in a similar manner. For example, the following:

```
<seealso cref="Employee(int, string)">
```

becomes this:

```
<seealso cref="M:Payroll.Employee.#ctor(System.Int32,System.String)">
```

The reference was to a constructor method that had an `int` as the first parameter and a `string` as the second parameter.

In addition to the preceding translations, the compiler wraps the XML information about each code element in a `member` tag that specifies the name of the member using the same encoding. This allows a post-processing tool to easily match up members and references to members.

The generated XML file from the preceding example is as follows (with a few word wraps):

```
<?xml version="1.0"?>
<doc>
    <assembly>
        <name>employee</name>
    </assembly>
    <members>
        <member name="T:Payroll.Employee">
            <summary>
            The Employee class holds data about an employee.
            This class contains a <see cref="T:System.String">string</see>
            </summary>
        </member>
        <member name="M:Payroll.Employee.#ctor(System.Int32,System.String)">
            <summary>
            Constructor for an Employee instance. Note that
            <paramref name="name2">name</paramref> is a string.
            </summary>
            <param name="id">Employee id number</param>
            <param name="name">Employee Name</param>
        </member>
        <member name="M:Payroll.Employee.#ctor">
            <summary>
            Parameterless constructor for an employee instance
            </summary>
            <remarks>
            <seealso cref="M:Payroll.Employee.#ctor(System.Int32,System.String)"
>Employee(int, string)</seealso>
            </remarks>
        </member>
    </members>
</doc>
```

The post-processing on a file can be quite simple; you can add an XSL file that specifies how the XML should be rendered, which leads to the display shown in Figure 38-1 (in a browser that supports XSL).

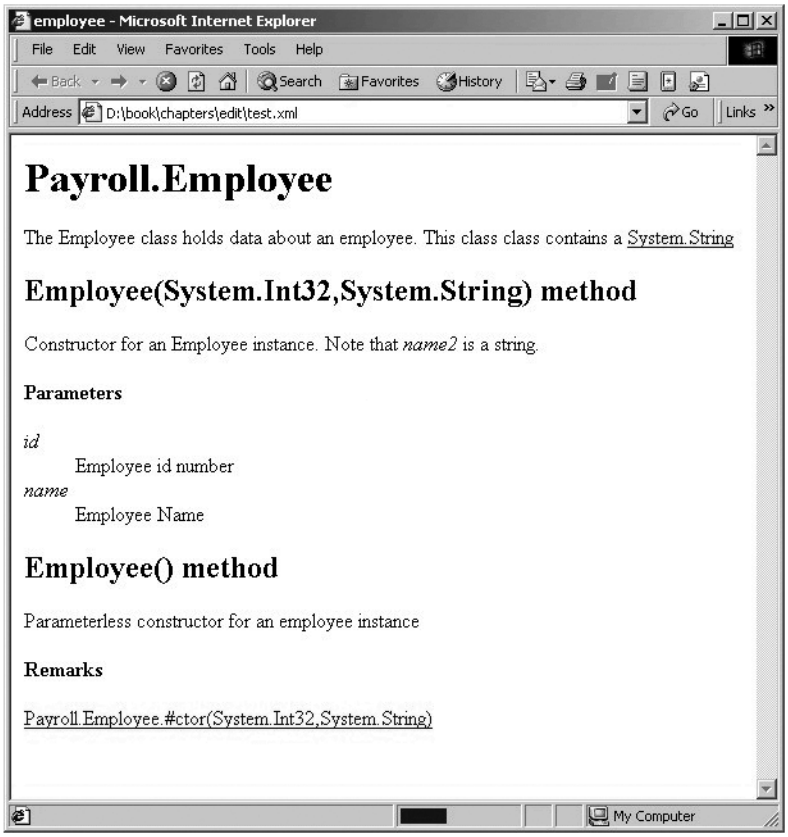

Figure 38-1. *XML file in Internet Explorer with formatting specified by an XSL file*

Note that XML documentation isn't designed to be a full solution to generating documentation. The final documentation should be generated by a tool that uses reflection to examine a class and combines the information from XML documentation to the information from reflection to produce the final documentation.

XML Documentation Tags

The remainder of the XML documentation tags describes the .NET documentation convention. They can be extended, modified, or ignored if necessary for a specific project. Table 38-2 shows the XML elements you can use for documentation.

Table 38-2. *XML Documentation Elements*

Tag	Description
<summary>	A short description of the item
<remarks>	A long description of the item
<c>	Formats characters as code within other text
<code>	Multiline section of code—usually used in an <example> section
<example>	An example of using a class or method
<exception>	The exceptions a class throws
<list>	A list of items
<param>	Describes a parameter to a member function
<paramref>	A reference to a parameter in other text
<permission>	The permission applied to a member
<returns>	The return value of a function
<see cref="member">	A link to a member or field in the current compilation environment
<seealso cref="member">	A link in the "see also" section of the documentation
<value>	Describes the value of a property

XML Include Files

In a project that has a separate technical-writing team, it may be more convenient to keep the XML text outside the code. To support this, C# provides an `include` syntax for XML documentation. Instead of having all the documentation before a function, you can use the following `include` statement:

```
/// <include file='Foo.csx' path='doc/member[@name="Foo.Comp"]' />
```

This will open the `Foo.csx` file and look for a `<doc>` tag. Inside the `doc` section, it will then look for a `<member>` tag that has the name `Foo.Comp` specified as an attribute. In other words, it will look for something like this:

```
<doc>
    <member name="Foo.Comp">
        <summary>A description of the routine</summary>
       <param name="obj1">the first object</param>
    </member>
    …
</doc>
```

Once the compiler has identified the matching section from the include file, it proceeds as if the XML were contained in the source file.

Garbage Collection in the .NET Runtime

Garbage collection has a bad reputation in a few areas of the software world. Some programmers think they can do a better job at memory allocation than a garbage collector (GC) can.

They're correct; they can do a better job but only with a custom allocator for each program and possibly for each class. Also, custom allocators are a lot of work to write, to understand, and to maintain.

In the vast majority of cases, a well-tuned garbage collector will give similar or better performance to an unmanaged heap allocator.

The following sections explain a bit about how the garbage collector works, how it can be controlled, and what can't be controlled in a garbage-collected world. The information presented describes the situation for platforms such as PCs and servers that run full versions of Windows. Systems with more constrained resources, such as the Pocket PC, are likely to have simpler GC systems.

Note also that there are optimizations performed for multiproc and server machines (covered later in the "Server vs. Workstation Garbage Collection" section).

Allocation

Heap allocation in the .NET runtime world is fast; all the system has to do is make sure the managed heap has enough room for the requested object, return a pointer to that memory, and increment the pointer to the end of the object.

Garbage collectors trade simplicity at allocation time for complexity at cleanup time. Allocations are really, really fast in most cases, though if there isn't enough room, a garbage collection might be required to obtain enough room for object allocation.

Of course, to make sure it has enough room, the system might have to perform a garbage collection.

To improve performance, large objects (greater than 20KB) are allocated from a large object heap.

Mark and Compact

The .NET garbage collector uses a Mark and Compact algorithm. When a collection is performed, the garbage collector starts at root objects (including globals, statics, locals, and CPU registers) and finds all the objects referenced from those root objects. This collection of objects denotes the objects that are in use at the time of the collection, and therefore all other objects in the system are no longer needed.

To finish the collection process, all the referenced objects are copied down in the managed heap, and the pointers to those objects are all fixed up. Then, the pointer for the next available spot is moved to the end of the referenced objects.

Since the garbage collector is moving objects and object references, no other operations can be going on in the system. In other words, all useful work must stop while the GC takes place.

Generations

It's costly to walk through all the objects that are currently referenced. Much of the work in doing this will be wasted work, since the older an object is, the more likely it is to stay. Conversely, the younger an object is, the more likely it is to be unreferenced.

The runtime capitalizes on this behavior by implementing generations in the garbage collector. It divides the objects in the heap into three generations:

- Generation 0 objects are newly allocated objects that have never been considered for collection.
- Generation 1 objects have survived a single garbage collection.
- Generation 2 objects have survived multiple garbage collections.

In design terms, generation 2 tends to contain long-lived objects, such as applications; generation 1 tends to contain objects with medium lifetimes, such as forms or lists; and generation 0 tends to contain short-lived objects, such as local variables.

When the runtime needs to perform a collection, it first performs a generation 0 collection. This generation contains the largest percentage of unreferenced objects and will therefore yield the most memory for the least work. If collecting that generation doesn't generate enough memory, generation 1 will then be collected and finally, if required, generation 2.

Figure 38-2 illustrates some objects allocated on the heap before a garbage collection takes place. The numerical suffix indicates the generation of the object; initially, all objects will be of generation 0. Active objects are the only ones shown on the heap, but space exists for additional objects to be allocated.

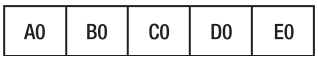

Figure 38-2. *Initial memory state before any garbage collection*

At the time of the first garbage collection, `B` and `D` are the only objects still in use. The heap looks like Figure 38-3 after collection.

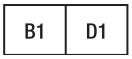

Figure 38-3. *Memory state after first garbage collection*

Since `B` and `D` survived a collection, their generation is incremented to 1. New objects are then allocated, as shown in Figure 38-4.

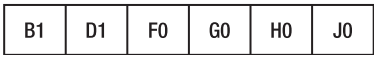

Figure 38-4. *New objects are allocated.*

Time passes. When another garbage collection occurs, `D`, `G`, and `H` are the live objects. The garbage collector tries a generation 0 collection, which leads to the layout shown in Figure 38-5.

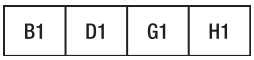

Figure 38-5. *Memory state after a generation 0 collection*

Even though `B` is no longer live, it doesn't get collected because the collection was for generation 0 only. After a few new objects are allocated, the heap looks like Figure 38-6.

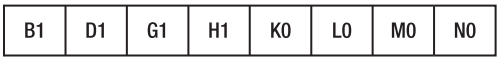

Figure 38-6. *More new objects are allocated.*

Time passes, and the live objects are `D`, `G`, and `L`. The next garbage collection does both generation 0 and generation 1, which leads to the layout shown in Figure 38-7.

Figure 38-7. *Memory state after a generation 0 and generation 1 garbage collection*

Finalization

The GC supports *finalization*, which is somewhat analogous to destructors in C++. In C#, they're known as *destructors* and are declared with the same syntax as C++ destructors, but from the runtime perspective, they're known as *finalizers*.

Finalizers allow the opportunity to perform some cleanup before an object is collected, but they have considerable limitations and therefore really shouldn't be used much.

Before we discuss their limitations, we'll describe how they work. When an object with a finalizer is allocated, the runtime adds the object reference to a list of objects that will need finalization. When a garbage collection occurs, if an object has no references but is contained on the finalization list, it's marked as ready for finalization.

After the garbage collection has completed, the finalizer thread wakes up and calls the finalizer for all objects that are ready for finalization. After the finalizer is called for an object, it's removed from the list of objects that need finalizers, which will make it available for collection the next time garbage collection occurs.

This scheme results in the following limitations regarding finalizers:

- Objects that have finalizers have more overhead in the system, and they hang around longer.
- Finalization takes place on a separate thread from execution.
- Finalization has no guaranteed order. If object `a` has a reference to object `b`, and both objects have finalizers, the object `b` finalizer might run before the object `a` finalizer, and therefore object `a` might not have a valid object `b` to use during finalization.

- Although finalizers are usually called on normal program exit, sometimes this won't occur. If a process is terminated aggressively (for example, if the Win32 `TerminateProcess` function is called), finalizers won't run. Finalizers can also fail to run if the finalization queue gets stuck running finalizers for a long time on process exit. In this case, attempts to run the finalizers will time out.

All of these limitations are why doing work in destructors is discouraged.

Controlling GC Behavior

At times, it may be useful to control the GC behavior. You should do this in moderation; the whole point of a managed environment is that it controls what's going on, and controlling it tightly can lead to problems elsewhere.

Forcing a Collection

The function `System.GC.Collect()` can be called to force a collection. Avoid this call in production code. Forcing a garbage collection is usually done as a futile attempt to reclaim resources that should have been disposed, and this problem has other remedies, such as using code-quality tools from companies such as Compuware that can pinpoint missed `Dispose()` calls.

Suppressing Finalization

As mentioned earlier, an instance of an object is placed on the finalization list when it's created. If it turns out that an object doesn't need to be finalized (because the cleanup function has been called, for example), you can use the `System.GC.SupressFinalize()` function to remove the object from the finalization list.

Server vs. Workstation Garbage Collection

The .NET Framework ships with two flavors of the garbage collection: server and workstation. The term *flavor* prevents confusion between the two different garbage collectors and different versions of the .NET Framework, such as 1.0, 1.1, and 2.0; we'll use this term for the rest of this discussion.

The workstation garbage collection is further broken down into two separate modes: concurrent and nonconcurrent. *Concurrent* garbage collection refers to the execution of managed code while various aspects of a garbage collection cycle are being completed. In the early description of the Mark and Compact algorithm, we stated that all useful work needs to be stopped while a garbage collection takes place. With concurrent garbage collection, this isn't quite true, because execution of managed code (in other words, useful work) can still occur during a couple of places during the collection cycle, notably during the Mark stage. Concurrent garbage collection makes for a slower overall garbage collection cycle, but during the collection the application stays more responsive. These characteristics make it the best choice for interactive applications such as Windows Forms applications, and this is the default mode for the garbage collector.

For some applications, such as those that operate in batch mode without a user interface, maximum performance in a garbage collection cycle is preferable to the minimization of pauses, and concurrent garbage collection should be turned off. If an unmanaged application written in a language such as C++ is explicitly hosting C# or other managed code, you can control the

use of concurrent garbage collection by the value of parameters passed to the hosting APIs (see the documentation for `CorBindToRuntimeEx` for details). For applications written entirely in managed code, you can use a configuration file setting to turn off concurrent garbage collection:

```
<configuration>
 <runtime>
  <gcConcurrent enabled="false"/>
 </runtime>
</configuration>
```

For a multiprocessor machine, many different optimizations are possible. To implement these optimizations, a different flavor of the garbage collector is available. The server flavor of the garbage collection will load only on multiprocessor machines, but it's possible to also load the workstation flavor on these machines, and this is actually the default behavior. Prior to Service Pack 1 (SP1) of the .NET Framework 1.1, the only supported[1] way to load the server flavor of the garbage collector was via the hosting APIs and the use of unmanaged code (again, see the documentation for `CorBindToRuntimeEx` for details). From .NET 1.1 SP1 onward, you can load the server flavor by using a configuration file setting:

```
<configuration>
 <runtime>
  <gcServer enabled="true"/>
 </runtime>
</configuration>
```

The server flavor of the garbage collector doesn't have a concurrent collection mode, and managed code is suspended for the duration of a collection cycle. This means that for UI-centric applications, the server version of the runtime may not give the best appearance of performing as well as the workstation flavor using concurrent collection. This is the reason for the default choice of concurrent workstation garbage collection regardless of the number of processors present.

For server applications, the optimizations present in the server flavor of the garbage collector can be quite significant. The main optimization is a separate garbage-collected heap and garbage collection thread for each processor. This dramatically reduces the amount of contention on global locks and resources needed to allocate and clean up memory. In the workstation flavor of the runtime, garbage collection takes places on the thread that requested the allocation that couldn't be satisfied from the currently available memory, which in turn triggered the garbage collection. In contrast, the optimizations in the server GC mean that every garbage-collected heap can be collected simultaneously, maximizing the use of the available processor power.

The 2.0 release of the .NET Framework libraries includes a new class called `GCSettings` that can determine which flavor of the garbage collection is in use:

```
bool isServerGC = GCSettings.IsServerGC;
```

1. You can use a Windows environment variable to instruct the CLR to load the server flavor of the garbage collector, but this was unsupported. To use this hack, create an environment variable called `COMPLUS_BUILDFLAVOR` and set it to `svr`.

Deeper Reflection

Examples in Chapter 22 showed how to use reflection to determine the attributes attached to a class. You can also use reflection to find all the types in an assembly or dynamically locate and call functions in an assembly. You can even use it to emit the .NET intermediate language on the fly to generate code that can be executed directly.

The documentation for the .NET CLR contains more details on using reflection.

Listing All the Types in an Assembly

The following example looks through an assembly and locates all the types in that assembly:

```
using System;
using System.Reflection;
enum MyEnum
{
    Val1,
    Val2,
    Val3
}
class MyClass
{
}
struct MyStruct
{
}
class Test
{
    public static void Main(String[] args)
    {
            // list all types in the assembly that is passed
            // in as a parameter
        Assembly a = Assembly.LoadFrom (args[0]);
        Type[] types = a.GetTypes();

            // look through each type, and write out some information
            // about them.
        foreach (Type t in types)
        {
            Console.WriteLine ("Name: {0}", t.FullName);
            Console.WriteLine ("Namespace: {0}", t.Namespace);
            Console.WriteLine ("Base Class: {0}", t.BaseType.FullName);
        }
    }
}
```

If this example runs, passing the name of the .exe in, it will generate the following output:

```
Name: MyEnum
Namespace:
Base Class: System.Enum
Name: MyClass
Namespace:
Base Class: System.Object
Name: MyStruct
Namespace:
Base Class: System.ValueType
Name: Test
Namespace:
Base Class: System.Object
```

Finding Members

This example lists the members of a type:

```
using System;
using System.Reflection;
class MyClass
{
    MyClass() {}
    static void Process()
    {
    }
    public int DoThatThing(int i, Decimal d, string[] args)
    {
        return(55);
    }
    public int        value = 0;
    public float        log = 1.0f;
    public static int    value2 = 44;
}
class Test
{
    public static void Main(String[] args)
    {
            // Iterate through the fields of the class
        Console.WriteLine("Fields of MyClass");
        type t = typeof (MyClass);
        foreach (MemberInfo m in t.GetFields())
        {
            Console.WriteLine("{0}", m);
        }
```

```
            // and iterate through the methods of the class
        Console.WriteLine("Methods of MyClass");
        foreach (MethodInfo m in t.GetMethods())
        {
            Console.WriteLine("{0}", m);
            foreach (ParameterInfo p in m.GetParameters())
            {
                Console.WriteLine("  Param: {0} {1}",
                    p.ParameterType, p.Name);
            }
        }
    }
}
```

This example produces the following output:

```
Fields of MyClass
Int32 value
Single log
Int32 value2
Methods of MyClass
Void Finalize ()
Int32 GetHashCode ()
Boolean Equals (System.Object)
  Param: System.Object obj
System.String ToString ()
Void Process ()
Int32 DoThatThing (Int32, System.Decimal, System.String[])
  Param: Int32 i
  Param: System.Decimal d
  Param: System.String[] args
System.Type GetType ()
System.Object MemberwiseClone ()
```

For information on how to reflect over an enum, see Chapter 21.

When iterating over the methods in `MyClass`, the standard methods from `object` also show up.

Invoking Functions

In this example, reflection will open the names of all the assemblies on the command lines to search for the classes in them that implement a specific interface and then to create an instance of those classes and invoke a function on the instance.

This is useful to provide a late-bound architecture, where a component can be integrated with other components' runtimes.

This example consists of four files. The first one defines the `IProcess` interface that will be searched for. The second and third files contain classes that implement this interface, and each is compiled to a separate assembly. The last file is the driver file; it opens the assemblies passed

on the command line and searches for classes that implement `IProcess`. When it finds one, it instantiates an instance of the class and calls the `Process()` function.

IProcess.cs

`IProcess` defines that interface that you'll search for:

```
// file=IProcess.cs
namespace MamaSoft
{
    interface IProcess
    {
        string Process(int param);
    }
}
```

process1.cs

This is the `process1.cs` file:

```
// file=process1.cs
// compile with: csc /target:library process1.cs iprocess.cs
using System;
namespace MamaSoft
{
    class Processor1: IProcess
    {
        Processor1() {}

        public string Process(int param)
        {
            Console.WriteLine("In Processor1.Process(): {0}", param);
            return("Raise the mainsail! ");
        }
    }
}
```

This should be compiled with the following:

```
csc /target:library process1.cs iprocess.cs
```

process2.cs

This is the `process2.cs` file:

```
// file=process2.cs
// compile with: csc /target:library process2.cs iprocess.cs
using System;
namespace MamaSoft
{
```

```
    class Processor2: IProcess
    {
        Processor2() {}

        public string Process(int param)
        {
            Console.WriteLine("In Processor2.Process(): {0}", param);
            return("Shiver me timbers! ");
        }
    }
}
class Unrelated
{
}
```

This should be compiled with the following:

```
csc /target:library process2.cs iprocess.cs
```

driver.cs

This is the `driver.cs` file:

```
// file=driver.cs
// compile with: csc driver.cs iprocess.cs
using System;
using System.Reflection;
using MamaSoft;
class Test
{
    public static void ProcessAssembly(string aname)
    {
        Console.WriteLine("Loading: {0}", aname);
        Assembly a = Assembly.LoadFrom (aname);

            // walk through each type in the assembly
        foreach (Type t in a.GetTypes())
        {
                // if it's a class, it might be one that we want.
            if (t.IsClass)
            {
                Console.WriteLine("  Found Class: {0}", t.FullName);

                    // check to see if it implements IProcess
                if (t.GetInterface("IProcess") == null)
                    continue;
```

```
                    // it implements IProcess. Create an instance
                       // of the object.
                object o = Activator.CreateInstance(t);

                    // create the parameter list, call it,
                    // and print out the return value.
                Console.WriteLine("    Calling Process() on {0}",
                           t.FullName);
                object[] args = new object[] {55};
                object result;
                result = t.InvokeMember("Process",
                    BindingFlags.Default |
                    BindingFlags.InvokeMethod,
                    null, o, args);
                Console.WriteLine("    Result: {0}", result);
            }
        }
    }
    public static void Main(String[] args)
    {
        foreach (string arg in args)
            ProcessAssembly(arg);
    }
}
```

After this sample has been compiled, it can be run with the following:

```
process process1.dll process2.dll
```

This generates the following output:

```
Loading: process1.dll
  Found Class: MamaSoft.Processor1
    Calling Process() on MamaSoft.Processor1
In Processor1.Process(): 55
    Result: Raise the mainsail!
Loading: process2.dll
  Found Class: MamaSoft.Processor2
    Calling Process() on MamaSoft.Processor2
In Processor2.Process(): 55
    Result: Shiver me timbers!
  Found Class: MamaSoft.Unrelated
```

For more information on generating code at execution time, see Chapter 32.

When calling functions with `MemberInvoke()`, any exceptions thrown will be wrapped in a `TargetInvocationException`, so the actual exception is accessed through the inner exception.

Dealing with Generics

Reflection has been fully updated to handle generics. Many new methods and properties deal with the potential that any given type or method parameter could be a generic type. The simplest way to determine if a particular type is generic is the new property of `Type` called `IsGenericTypeDefinition`. This property will return true only if the type is generic and the generic types haven't been bound to a nongeneric type:

```
class Program
{
 static void Main(string[] args)
 {
   List<int> l = new List<int>();

   //will be false
   bool b1 = l.GetType().IsGenericTypeDefinition;

   //will be true
   bool b2 = l.GetType().GetGenericTypeDefinition().IsGenericTypeDefinition;
 }
}
```

In this case, the `IsGenericTypeDefinition` returns false for the type `List<int>`, which is a type that doesn't have any generic parameters. You can use the method `GetGenericTypeDefinition()` to get a reference from the constructed type `List<int>` back to the unbound generic type `List<T>`. The `IsGenericTypeDefinition` property returns true for this unbound generic type.

You can access the generic arguments for a type or method via the `GetGenericArguments()` method. Consider the following generic type:

```
class MyGenericClass<T> { }
```

You can display the generic parameter with the following code:

```
static void DumpGenericTypeParams(Type t)
{
 if (t.IsGenericTypeDefinition)
 {
  foreach (Type genericType in t.GetGenericArguments())
  {
   Console.WriteLine(genericType.Name);
  }
 }
}
```

The output from this code when run against `MyGenericClass<T>` is simply as follows:

```
T
```

Although simply dumping out the type name may not be overly useful, various reflection methods exist to access information such as the constraints that apply to the generic parameters (`Type.GetGenericParameterConstraints()`) and to bind generic parameters to nongeneric types (`Type.BindGenericParameters()`).

Optimizations

The C# compiler performs the following optimizations when the `/optimize+` flag is used:

- Local variables that are never read are eliminated (even if they're assigned to).
- Unreachable code (code after a `return`, for example) is eliminated.
- A `try-catch` with an empty `try` block is eliminated.
- A `try-finally` with an empty `try` is converted to normal code.
- A `try-finally` with an empty `finally` is converted to normal code.
- Branch optimization is performed.
- Field initializers that set a member variable to its default value are removed; this optimization, new in the 2.0 release of the compiler, means there's no performance penalty for setting integer values to `0`, Booleans to `false`, and references to `null`. Any code that does this is optimized, and the compiler relies on the CLR to intialize these fields to their correct value.

Additionally, when optimization is turned on, it enables optimizations by the JIT compiler.

CHAPTER 39

Defensive Programming

The .NET runtime provides a few facilities for making programming less dangerous. You can use conditional methods and tracing to add checks and log code to an application, to catch errors during development, and to diagnose errors in released code.

Conditional Methods

Conditional methods are typically used to write code that performs operations only when compiled in a certain way. This often takes place in order to add code that's called only when a debug build is made and not when called in other builds, usually because the additional check is too slow.

In C++, you'd do this by using a macro in the `include` file that changes a function call to nothing if the debug symbol isn't defined. This doesn't work in C#, however, because there's no `include` file or macro.

In C#, you can mark a method with the `Conditional` attribute to indicate when calls to it should be generated. For example:

```
using System;
using System.Diagnostics;

class MyClass
{
    public MyClass(int i)
    {
        this.i = i;
    }

    [Conditional("DEBUG")]
    public void VerifyState()
    {
        if (i != 0)
            Console.WriteLine("Bad State");
    }
```

```
    int i = 0;
}

class Test
{
    public static void Main()
    {
        MyClass c = new MyClass(1);

        c.VerifyState();
    }
}
```

The `VerifyState()` function has the `Conditional` attribute applied to it, with `DEBUG` as the conditional string. When the compiler comes across a function call to such a function, it looks to see if the conditional string has been defined. If it hasn't been defined, the call to the function is eliminated.

If this code is compiled using `/D:DEBUG` on the command line, it will print `Bad State` when it runs. If it's compiled without `DEBUG` defined, the function won't be called, and there will be no output.

Debug and Trace Classes

The .NET runtime has generalized this concept by providing the `Debug` and `Trace` classes in the `System.Diagnostics` namespace. These classes implement the same functionality but have slightly different uses. Code that uses the `Trace` classes is intended to be present in released software, and therefore it's important not to overuse it, as it could affect performance.

`Debug`, on the other hand, isn't going to be present in the released software; therefore, you can use it more liberally.

Calls to `Debug` are conditional on `DEBUG` being defined, and calls to `Trace` are conditional on `TRACE` being defined. By default, the Visual Studio IDE will define `TRACE` on both debug and retail builds and will define `DEBUG` only on debug builds. When compiling from the command line, you'll need the appropriate option.

In the remainder of this chapter, examples that use `Debug` also work with `Trace`.

Asserts

An *assert* is simply a statement of a condition that should be true, followed by some text to output if it's false. You could write the preceding code example better as this:

```
// compile with: csc /r:system.dll file_1.cs
using System;
using System.Diagnostics;
```

```
class MyClass
{
    public MyClass(int i)
    {
        this.i = i;
    }

    [Conditional("DEBUG")]
    public void VerifyState()
    {
        Debug.Assert(i == 0, "Bad State");
    }

    int i = 0;
}

class Test
{
    public static void Main()
    {
        Debug.Listeners.Clear();
        Debug.Listeners.Add(new TextWriterTraceListener(Console.Out));
        MyClass c = new MyClass(1);

        c.VerifyState();
    }
}
```

By default, asserts and other debug output are sent to all the listeners in the `Debug.Listeners` collection. Since the default behavior is to open a dialog box, the code in `Main()` clears the `Listeners` collection and then adds a new listener that's hooked up to `Console.Out`. This results in the output going to the console.

Asserts are hugely useful in complex projects to ensure that expected conditions are true.

Debug and Trace Output

In addition to using asserts, you can use the `Debug` and `Trace` classes to send useful information to the current debug or trace listeners. This is a useful adjunct to running in the debugger, because it's less intrusive and can be enabled in released builds to generate log files.

The `Write()` and `WriteLine()` functions send output to the current listeners. These are useful in debugging but not really useful in released software, since it's rare to want to log something all the time.

The `WriteIf()` and `WriteLineIf()` functions send output only if the first parameter is true. This allows the behavior to be controlled by a static variable in the class, which you could change at runtime to control the amount of logging that's performed. For example:

```
// compile with: csc /r:system.dll file_1.cs
using System;
using System.Diagnostics;
class MyClass
{
    public MyClass(int i)
    {
        this.i = i;
    }

    [Conditional("DEBUG")]
    public void VerifyState()
    {
        Debug.WriteLineIf(debugOutput, "In VerifyState");
        Debug.Assert(i == 0, "Bad State");
    }

    static public bool DebugOutput
    {
        get
        {
            return(debugOutput);
        }
        set
        {
            debugOutput = value;
        }
    }

    int i = 0;
    static bool debugOutput = false;
}

class Test
{
    public static void Main()
    {
        Debug.Listeners.Clear();
        Debug.Listeners.Add(new TextWriterTraceListener(Console.Out));
        MyClass c = new MyClass(1);

        c.VerifyState();
        MyClass.DebugOutput = true;
        c.VerifyState();
    }
}
```

This code produces the following output:

```
Fail: Bad State
In VerifyState
Fail: Bad State
```

Using Switches to Control Debug and Trace

The previous example showed how to control logging based upon a `bool` variable. The drawback of this approach is that there must be a way to set that variable within the program. It'd be more useful to have a way to set the value of such a variable externally.

The `BooleanSwitch` and `TraceSwitch` classes provide this feature. You can control their behavior at runtime by either setting an environment variable or setting a registry entry.

BooleanSwitch

The `BooleanSwitch` class encapsulates a simple Boolean variable, which is then used to control logging:

```
// file=boolean.cs
// compile with: csc /D:DEBUG /r:system.dll boolean.cs
using System;
using System.Diagnostics;

class MyClass
{
    public MyClass(int i)
    {
        this.i = i;
    }

    [Conditional("DEBUG")]
    public void VerifyState()
    {
        Debug.WriteLineIf(debugOutput.Enabled, "VerifyState Start");

        if (debugOutput.Enabled)
            Debug.WriteLine("VerifyState End");
    }

    BooleanSwitch    debugOutput =
            new BooleanSwitch("MyClassDebugOutput", "Control debug output");
    int i = 0;
}
```

```
class Test
{
    public static void Main()
    {
        Debug.Listeners.Clear();
        Debug.Listeners.Add(new TextWriterTraceListener(Console.Out));
        MyClass c = new MyClass(1);

        c.VerifyState();
    }
}
```

This example creates an instance of `BooleanSwitch` as a static member of the class, and this variable controls whether output happens. If this code runs, it produces no output, but you can control the `debugOutput` variable by setting the value in the configuration file for the assembly. This file is named `<assembly-name>.config`, which for this example means it's called `boolean.exe.config`, and it has to be in the same directory as the assembly. Not surprisingly, the config file uses XML to store its values. Here's the config file for the example:

```
<configuration>
    <system.diagnostics>
       <switches>
          <add name="MyClassDebugOutput" value="1" />
       </switches>
    </system.diagnostics>
 </configuration>
```

Running the code using this file produces the following results:

```
VerifyState Start
VerifyState End
```

The code in `VerifyState` shows two ways of using the variable to control output. The first usage passes the flag off to the `WriteLineIf()` function and is the simpler one to write. It's a bit less efficient, however, since the function call to `WriteLineIf()` is made even if the variable is false. The second version, which tests the variable before the call, avoids the function call and is therefore slightly more efficient.

TraceSwitch

It's sometimes helpful to use something other than a Boolean to control logging. It's common to have different logging levels, each of which writes a different amount of information to the log.

The TraceSwitch class defines five levels of information logging. They're defined in the TraceLevel enum. Table 39-1 shows the possible values of TraceLevel.

Table 39-1. *TraceLevel Values*

Level	Numeric Value
Off	0
Error	1
Warning	2
Info	3
Verbose	4

Each of the higher levels implies the lower level; for example, if the level is set to Info, Error and Warning will also be set. You use these numeric values when setting the flag via an environment variable or registry setting.

The TraceSwitch class exposes properties that tell whether a specific trace level has been set, and a typical logging statement checks to see whether the appropriate property was set. Here's the previous example, modified to use different logging levels:

```
// compile with: csc /r:system.dll file_1.cs
using System;
using System.Diagnostics;

class MyClass
{
    public MyClass(int i)
    {
        this.i = i;
    }

    [Conditional("DEBUG")]
    public void VerifyState()
    {
        Debug.WriteLineIf(debugOutput.TraceInfo, "VerifyState Start");

        Debug.WriteLineIf(debugOutput.TraceVerbose,
            "Starting field verification");

        if (debugOutput.TraceInfo)
            Debug.WriteLine("VerifyState End");
    }
```

```
    static TraceSwitch    debugOutput =
        new TraceSwitch("MyClassDebugOutput", "Control debug output");
    int i = 0;
}

class Test
{
    public static void Main()
    {
        Debug.Listeners.Clear();
        Debug.Listeners.Add(new TextWriterTraceListener(Console.Out));
        MyClass c = new MyClass(1);

        c.VerifyState();
    }
}
```

User-Defined Switch

The `Switch` class nicely encapsulates getting the switch value from the registry, so it's easy to derive a custom switch if the values of `TraceSwitch` don't work well.

The following example implements `SpecialSwitch`, which implements the `Mute`, `Terse`, `Verbose`, and `Chatty` logging levels:

```
// compile with: csc /r:system.dll file_1.cs
using System;
using System.Diagnostics;

enum SpecialSwitchLevel
{
    Mute = 0,
    Terse = 1,
    Verbose = 2,
    Chatty = 3
}

class SpecialSwitch: Switch
{
    public SpecialSwitch(string displayName, string description) :
        base(displayName, description)
    {
    }
```

```
public SpecialSwitchLevel Level
{
    get
    {
       return((SpecialSwitchLevel) base.SwitchSetting);
    }
    set
    {
        base.SwitchSetting = (int) value;
    }
}
public bool Mute
{
    get
    {
        return(base.SwitchSetting == 0);
    }
}

public bool Terse
{
    get
    {
        return(base.SwitchSetting  >= (int) (SpecialSwitchLevel.Terse));
    }
}
public bool Verbose
{
    get
    {
        return(base.SwitchSetting  >= (int) SpecialSwitchLevel.Verbose);
    }
}
public bool Chatty
{
    get
    {
        return(base.SwitchSetting  >=(int) SpecialSwitchLevel.Chatty);
    }
}
```

```
     protected new int SwitchSetting
    {
        get
        {
            return((int) base.SwitchSetting);
        }
        set
        {
            if (value < 0)
                value = 0;
            if (value > 4)
                value = 4;

            base.SwitchSetting = value;
        }
    }

}

class MyClass
{
    public MyClass(int i)
    {
        this.i = i;
    }

    [Conditional("DEBUG")]
    public void VerifyState()
    {
    Console.WriteLine("VerifyState");
        Debug.WriteLineIf(debugOutput.Terse, "VerifyState Start");

        Debug.WriteLineIf(debugOutput.Chatty,
            "Starting field verification");

        if (debugOutput.Verbose)
            Debug.WriteLine("VerifyState End");
    }

    static SpecialSwitch    debugOutput =
        new SpecialSwitch("MyClassDebugOutput", "application");
    int i = 0;
}
```

```
class Test
{
    public static void Main()
    {
        Debug.Listeners.Clear();
        Debug.Listeners.Add(new TextWriterTraceListener(Console.Out));
        MyClass c = new MyClass(1);

        c.VerifyState();
    }
}
```

You can control this switch with the same config file as the previous example.

Capturing Process Metadata

Simply outputting a string that records what a particular thread is doing is insufficient to monitor operations and track down problems in production environments. It isn't usually necessary to have thousands of threads running within hundreds of processes on dozens of machines. Although it's interesting that a particular thread called `Trace.WriteLine()` exists and has recorded that an error occurred, finding the actual thread that originated the statement, along with other metadata about the state of the process, is also interesting. To allow you to capture this information without having to manually code its retrieval, the `Trace` class in the 2.0 release of the .NET Framework libraries includes a new method called `TraceInformation()`.

This method is capable of outputting the thread's call stack, the date and time the `Trace` statement was made, the logical operation stack (which is the chain of calls in the current call context and may span multiple threads), the process ID, the thread ID, and the time stamp (which is a `long` that represents the number of ticks in the system timer and is a higher resolution than the date and time value that's also available). The actual listener decides which output values should be collected for a `TraceInformation` statement. To add and output all the available information, use the following code:

```
 ConsoleTraceListener ctl = new ConsoleTraceListener();
ctl.TraceOutputOptions = TraceOptions.Callstack | TraceOptions.DateTime |
 TraceOptions.LogicalOperationStack | TraceOptions.ProcessId |
 TraceOptions.ThreadId | TraceOptions.Timestamp;
System.Diagnostics.Trace.Listeners.Add(ctl);
Trace.TraceInformation("An error occurred ");
```

Running this code produces the following output:

```
Trace.exe Information: 0 : An error occurred
    ProcessId=2324
    LogicalOperationStack=
    ThreadId=1
    DateTime=2005-01-25T10:52:56.4135000Z
    Timestamp=15259908099
       Callstack=   at System.Environment.GetStackTrace(Exception e,
       Boolean needFileInfo)
   at System.Environment.get_StackTrace()
   at System.Diagnostics.TraceEventCache.get_Callstack()
   at System.Diagnostics.TraceListener.WriteFooter(TraceEventCache eventCache)
   at System.Diagnostics.TraceListener.TraceEvent(TraceEventCache eventCache,
    String source, TraceEventType severity, Int32 id, String message)
   at System.Diagnostics.TraceInternal.TraceEvent(TraceEventType severity,
   Int32 id, String format, Object[] args)
   at System.Diagnostics.Trace.TraceInformation(String message)
   at Trace1.Program.Main(String[] args) in
z:\nick\pic#\code\trace\trace\program.cs:line 17
```

Obviously, collecting all this information is an expensive process, and the output information should be collected during a `TraceInformation` call only as it's needed.

CHAPTER 40

Tips for Real-World Code

This chapter discusses the techniques for writing code in a manner that reduces the chances of bugs and delivers high-performance, reliable assemblies suitable for production use. These techniques reflect our cumulative experience (as an original C# team member in Eric Gunnerson's case and as a professional C# developer, consultant, and MVP in Nick Wienholt's case).

Some of the material in this chapter reinforces the advice presented in earlier chapters, and other sections cover new topics.

Naming Conventions

Chapter 38 listed the naming conventions you should use in C# applications. These recommendations are a subset of the full set of guidelines at `http://msdn.microsoft.com/library/default.asp?url=/library/en-us/cpgenref/html/cpconnamespacenamingguidelines.asp`, which in turn are a subset of Microsoft's own internal coding guidelines, which are currently (in early 2005) available at `http://blogs.msdn.com/brada/articles/361363.aspx`.

Note The first sentence of the documentation starts with "First, read the .NET Framework Design Guidelines," which may be helpful if you need to find the conventions using a Web search engine at some point in the future.

Naming conventions are primarily designed to speed up coding and to reduce bugs by allowing a developer to differentiate items such as types, variables, enumerations, and other code constructs based on the way they're named. Naming conventions work in conjunction with the language, the compiler, and the development environment, and advances in these areas can lessen the need for complex naming conventions. .NET and the C# language, which are strongly typed and have excellent facilities such as reflection for dealing with type information, make Hungarian notation redundant. (Hungarian notation uses type-identifying prefixes in variable names.)

Naming conventions are a means to an end (in this case, the end is faster, better code); they aren't an end goal in themselves. Because they're easy to learn and enforce, intermediate-level developers often go overboard in developing and enforcing naming standards and code layout rules that take a long time to implement and maintain. Avoid this tendency to overconventionalize

the way code is written, and realize that a few, clear rules are much more likely to be respected than a large, multivolume developer handbook.

The following tips will make living with naming conventions and coding guidelines a little easier:

Use the minimal set of naming conventions and coding guidelines necessary to arrive at good code. Chapter 38 will serve as a good basis.

Use coding guidelines (such as the position of braces and tabs/spaces) that are supported by the code editor. Visual Studio 2005 offers a great range of flexibility regarding these issues. Ensure that each developer is using the same settings. With the development environment set up correctly, you'll automatically take care of code layout issues. Although it's possible for each developer to maintain separate settings and use Visual Studio's reformatting power to modify source files for their edits, it will cause unneeded formatting discrepancies in the code base and confusing churn in the source control system.

For developers moving to .NET, use an auditing tool to pick up lapses in naming conventions. FxCop from Microsoft is built specifically for this purpose and is integrated into Visual Studio 2005.

Don't use Hungarian notation in non-UI code, even though Hungarian notation may still be useful in UI code to differentiate between the name of member variables and the UI element that displays them, as well as making controls of the same class easier to find using IntelliSense.

Embrace the IDE

A lot of real-world code does mundane and tedious work—managing object population, getting the information from objects into UI controls and back again, laying out screens, and dealing with the flow of data into and out of a database. Microsoft has expended an enormous sum of money in research and development to make Visual Studio .NET an indispensable aid for creating this tedious code in a fast and robust manner. If you're still an old-style developer cutting code in Notepad, try Visual Studio .NET 2005 for a month or so, making a genuine attempt to use its features to simplify your coding efforts. It's unlikely you'll choose to go back to Notepad at the end of the evaluation.

Exceptions

Chapter 4 introduced the topic of exception handling. The following sections summarize a number of the points made in Chapter 4 and also cover a few items not covered earlier.

Throw Early, Throw Often

To borrow from the famous electoral fraud campaign associated with Richard Daley (Chicago's Democratic mayor during the 1960 presidential election), it's good practice to throw exceptions early and often. Not only does this help with code correctness, it helps reduce the likelihood of security vulnerabilities. Noted Microsoft security expert Michael Howard has

stated that if security problems caused by uncaught malicious input could be solved, the vast majority of computer security issues would disappear (Microsoft Professional Developer Conference 2003, Security Symposium, Session 1).

Thoroughly checking all parameters that come into a method and rejecting the invalid parameters by throwing an `ArgumentException`-derived object is the best way to implement malicious input checks. The .NET Framework libraries are written using this technique (it's possible to verify this claim by using Reflector (`http://www.aisto.com/roeder/dotnet/`) or the Shared Source CLI (which is based on the commercial CLR and is available from `http://msdn.microsoft.com/net/sscli/`).

The actual pattern used is as follows:

```
using System;

class SomeRandomClass
{
 private static readonly int MaxValueP1 = 10;

 public void SomeRandomMethod(int p1, string p2)
 {
  if (p1 < 0 || p1 > SomeRandomClass.MaxValueP1)
  {
   throw new ArgumentOutOfRangeException("p1", p1,
    "p1 must be a positive number less than " + MaxValueP1.ToString());
  }

  if (p2 == null)
  {
   throw new ArgumentNullException("p2");
  }

  if (p2.Length > p1)
  {
   throw new ArgumentOutOfRangeException(
    "The length of p2 cannot be greater than the value of p1");
  }

  //real method body
 }

 //other methods
}
```

Although one of the guidelines related to exception management is to minimize the number of exceptions thrown, it's more important to avoid security bugs and any other type of bug rather than to incur the performance hit of exceptions. If performance is favored too highly over security, the code produced may end up as part of a fast-spreading Internet worm, and performance of this kind is in nobody's interests.

Catching, Rethrowing, and Ignoring Exceptions

This section serves only to reiterate the advice presented in Chapter 4. Many developers get these items wrong when developing their code.

The two simple rules are as follows:

If an exception can't be handled, don't catch it. Many developers, particularly those from a language background that doesn't include exceptions, feel obliged to catch every exception. Letting an exception that can't be handled pass through isn't a case of bad coding manners and doesn't indicate a lazy program. The stack trace will let the ultimate catcher know that the exception came through a particular method, so catching just to log is largely redundant. The only time when a global `catch` block is appropriate is at application or service boundaries. If resources need to be released, use a `finally` block rather than a `catch` block. The C# `using` statement (covered in a moment) simplifies this pattern.

Use `throw` instead of `throw ex` (where `ex` is the exception variable from the `catch` statement). Using `throw ex` will reset the stack trace, which is rarely the intended result. If code is rethrowing lots of exceptions, consider whether it's a violation of the first guideline.

Use using

When instantiating any scope-limited `IDisposable`-implementing object, use a `using` block to ensure the resource is properly released. A `using` block guarantees that a resource will be cleaned up properly—provided that the machine, process, or application domain hosting the code isn't forcibly shut down. If an exception is thrown, or a `return` statement is executed, the `finally` block generated by the C# compiler will execute, and the `IDisposable`-implementing object will have its `Dispose` method called if the object has been created.

You can include two objects in the same `using` statement if they're of the same type. If the objects are of different types, you can nest the `using` blocks:

```
using (SqlConnection conn1 = new SqlConnection(), conn2 = new SqlConnection())
{
 //some more code here
 using (SqlCommand cmd = new SqlCommand())
 {
 ;
 }
}
```

Collections

When upgrading from .NET 1.*x* to 2.0, the key challenge is to "reset your defaults" and move rapidly to the generic-based collections. Generics are CLS-compliant, which means all other .NET 2.0–compliant languages should at least be able to consume generic code, if not extend it. Generics have many advantages and essentially no disadvantages. The only two problems are developer familiarity and the requirement for end users to upgrade their .NET Framework version to 2.0. For the sake of a day of training and 20MB of hard disk space, the benefits of generics far outweigh this potential inconvenience.

Thread-Safety

Dealing with threads is one of the most challenging aspects typically facing an intermediate-level developer. At some stage, most developers "discover threads" and go wild with all sorts of weird and wonderful threading patterns. This usually lasts until they spend all night or all weekend in front of the computer attempting to diagnose some intermittent threading bug. The pain of this episode is useful in teaching a developer to use threads sparingly and only when appropriate. If you have yet to experience this event, conduct the mental exercise of placing yourself in the position of all the developers who have experienced it and attempt to learn from their mistakes.

As a general rule, apply threads only in the following situations:

- When a lengthy operation needs to occur and the UI thread must remain responsive to user input. Remember that Windows Forms Control methods and properties (with the exception of `IsInvokeRequired` and `Invoke()`) need to be invoked back onto the UI thread.
- When two or more operations need to be conducted and minimal dependencies exist between the two operations. Examples include servicing multiple clients simultaneously and writing some data to a database and flat file at the same time.

Understand Processor Memory Models

Chapter 31 touched on processor memory models, and, because this an introductory book, this section won't drill into the topic in much more detail. The message worth emphasizing from Chapter 31 is that it isn't wise to get too clever with threading optimizations until you understand the whole range of operating system, .NET Framework, and processor optimizations.

With the x86 instruction set and current 32-bit processors, the physical and logical execution of code paths is fairly similar. With the new range of 64-bit processors, logical and physical code paths begin to diverge, and a processor may do something such as speculatively execute a code branch or load a bit of data based on a guess that the code branch result or data will probably be needed. If the processor guesses wrong, the speculative results are discarded silently, but if an activity is occurring on another thread *without the normal locking statements*, you can get strange results. Normal locking statements are often not used with operations that are atomic at an instruction-set level, such as variable assignment. With more permissive memory models such as those on the 64-bit chips, this can cause problems. A specific processor instruction and a .NET Framework method (`Thread.MemoryBarrier()`) can deal with this problem without resorting to locking, but unless code is being written (and will be maintained) by developers with a mastery of all these issues, it's much wiser to take the small performance hit of locking.

Locking on this and Type

In .NET 1.*x*, the pattern of locking on an object's `this` reference for instance data and a type's `Type` instance for static data was the dominant pattern. This pattern has been deprecated in favor of holding private `object` references that are used with `Monitor` locks and the `lock` statement.

Using private variables prevents two separate pieces of code from locking on the same object for different purposes, which may potentially cause deadlock, and it prevents malicious code that's loaded through plug-ins or similar means from conducting Denial of Service attacks through thread locking.

The issues surrounding the use of locks on the `this` reference are a microcosm of the larger problem of ensuring that multiple threads don't interfere with each other in unintended and undesirable ways. Although there's a wealth of methods, ranging from the lightweight `Interlock` functions all the way up to heavyweight semaphores and mutexes, it's critical that the activity of various threads is partitioned enough so they aren't constantly trying to acquire locks on shared resources. If multiple threads are spending the majority of their time attempting to use shared resources to complete a unit of work, it could indicate certain design problems.

Code-Quality Tools

Developers have relied on code-quality tools such as PC-Lint to supplement the checks made by the compiler for many years. Although the strong typing and verification features of .NET and C# mean that many of the tasks that have traditionally been implemented by code-quality tools are now part of the compile process, the need for code-quality tools hasn't disappeared. The following sections cover a few entry-level tools that will be useful to developers getting started with .NET.

NUnit

Although Extreme Programming (XP) has failed to take over the world as the dominant software development methodology, agile programming's promotion of continuous integration and unit testing has had a marked impact on a huge range of developers and development shops that haven't adopted the other elements of XP. While unit testing and continuous integration weren't invented by XP, it has promoted them from a "should do" to a "must do" in the mind of most developers. This section focuses on unit testing; to learn more about continuous integration, consult one of the multitude of XP and development methodology books that have been released in the past five years.

Unlike in other areas of open-source tools, NUnit is the dominant force in .NET unit testing and is freely available from `http://www.nunit.org`. NUnit ships with an easy-to-read primer to get started with unit testing, so it'd be redundant to repeat that material here.

Writing unit tests is easy from a syntax perspective. The hardest part of unit testing is constructing unit tests that are meaningful enough so they're likely to trap bugs without being so tightly coupled to the implementation of a method that any change causes the test to fail. Although this is trivial to implement and demonstrate in a unit test that examines code that performs some basic arithmetic operation, it's much harder to achieve with real-world methods that have database and UI interactions. Even if you achieve loose coupling between components, testing application-level components such as `Employee` or `PayrollRun` can be a difficult task.

One of the best sources for looking at the way real-world unit tests are written is the Shared Source CLI (SSCLI) from Microsoft. While not written using NUnit (which obviously doesn't predate the .NET Framework), the tests compromise hundreds of real unit tests that helped ensure the high quality of the .NET Framework. The range and breadth of tests disprove the notion that "real" developers on "real" projects don't unit test and are a fantastic source of unit testing inspiration.

FxCop

FxCop statically analyzes a compiled assembly for compliance with .NET Framework guidelines. Microsoft originally created the tool to ensure the .NET Framework complied with these guidelines and provided a consistent programming experience. Microsoft released the tool as a free download as part of .NET 1.*x* (see the FxCop home page at `http://www.gotdotnet.com/team/fxcop/` for details) and has now integrated it into Visual Studio .NET 2005. Although FxCop can't enforce coding layout standards (it inspects only the compiled assembly), it can ensure compliance with naming guidelines and has many other security and efficiency checks.

As well as providing a good out-of-the-box experience for checking compiled assemblies, FxCop offers a great extension mechanism that allows assemblies to be checked without dealing with all the pain associated with loading and parsing MSIL. Although not as powerful and mature as third-party offerings from companies such as Compuware, the fact that FxCop is freely available makes it a good entry point into the code/assembly analysis world.

CHAPTER 41

The Command-Line Compiler

This chapter describes the command-line switches that you can pass to the compiler. Options that can be abbreviated are shown with the abbreviated portion in brackets ([]).

You can use the `/out` and `/target` options more than once in a single compilation, and they apply only to those source files that follow the option.

Simple Usage

In simple usage, you might use the following command:

```
csc test.cs
```

This compiles the file `test.cs` and produces a console assembly (`test.exe`) that can then be executed. You can specify multiple files on the same line, along with wildcards.

Response Files

The C# compiler supports a response file that contains command-line options. This is especially useful if you have lots of files to compile or complex options.

You specify a response file merely by listing it on the command line:

```
csc @<responsefile>
```

You can use multiple response files on a single command line, or you can mix them with options on the command line.

Default Response File

To avoid having to specify lots of assemblies, the compiler looks for a `csc.rsp` file, unless it's specifically told not to do so. If it finds one in the current directory, it will use that file as if it had been specified with the at (@) sign syntax. If the file doesn't exist in the current directory, it will next look in the directory where `csc.exe` lives.

Command-Line Options

The following tables summarize the command-line options for the C# compiler. Specifically, Table 41-1 specifies the error-reporting options, Table 41-2 describes the input options, Table 41-3 lists the output options, Table 41-4 specifies the processing options, and Table 41-5 shows miscellaneous options. You can also set most of these from within the Visual Studio IDE.

Table 41-1. *Error-Reporting Options*

Command	Description
`/warnaserror[+|-]`	Treats warnings as errors. When this option is on, the compiler will return an error code even if there were only warnings during the compilation. A plus (+) sign turns the option on, and a minus (-) sign turns the option off.
`/w[arn]:<level>`	Sets warning level (0–4).
`/nowarn:<list>`	Specifies a comma-separated list of warnings to not report.

Table 41-2. *Input Options*

Command	Description
`/addmodule:<file>`	Specifies modules that are to become part of this assembly.
`/codepage:<id>`	Uses the specified code page's ID to open source files.
`/nostdlib[+|-]`	Doesn't import the standard library (`mscorlib.dll`). You could use this to switch to a different standard library for a specific target device.
`/recurse:<filespec>`	Searches subdirectories for files to compile.
`/lib:<directory list>`	Specifies additional directories to search for references.
`/noconfig`	Doesn't include the `csc.rsp` file automatically.
`/r[eference]:<file>`	Specifies a metadata file to import.
`/reference <alias> = <file>`	Specifies a metadata file to import using an alias. See Chapter 11 for more details.

Table 41-3. *Output Options*

Command	Description
`/o[ptimize] [+|-]`	Enables optimizations.
`/out:<outfile>`	Sets output filename.
`/t[arget]:module`	Creates a module that can be added to another assembly.
`/t[arget]:library`	Creates a library instead of an application.

Table 41-3. *Output Options (Continued)*

Command	Description
/t[arget]:exe	Creates a console application (default).
/t[arget]:winexe	Creates a Windows GUI application.
/filealign:n	Specifies the alignment used for output file sections.
/delaysign	Signs the assembly using only the public portion of the strong name key. See the MSDN documentation for further details on strong naming.
/keyfile:<file>	Specifies a strong name key file.
/keycontainer:<string>	Specifies a strong name key container.
/platform:<string>	Limits which platforms this code can run on: x86, Itanium, x64, or anycpu. The default is anycpu.
/baseaddress:<addr>	Specifies the library base address.

Table 41-4. *Processing Options*

Command	Description
/debug:{full|pdbonly}	Emits debugging information. Fully allows connecting to a running instance.
/incr[emental] [+|-]	Performs an incremental build.
/checked[+|-]	Checks for overflow and underflow by default.
/unsafe[+|-]	Allows "unsafe" code.
/d[efine]:<def-list>	Defines conditional compilation symbol(s).
/doc:<file>	Specifies a file in which to store XML documentation comments.
/win32res:<resfile>	Specifies a Win32 resource file.
/win32icon:<iconfile>	Specifies a Win32 icon file.
/res[ource]:<file>[,<name>[,<MIMEtype]]	Embeds a resource into this assembly.
/langversion:<string>	Specifies language version modes: ISO-1 or Default. Using ISO-1 will restrict language features to those present in the first C# ISO standard and will prevent the use of features such as generics. Default will target the current language version.
/linkres[ource] :<file>[,<name>[,<MIMEtype]]	Links a resource into this assembly without embedding it.

Table 41-5. *Miscellaneous*

Command	Description
/? or /help	Displays the usage message.
/nologo	Doesn't display the compiler copyright banner.
/bugreport:<file>	Creates report file.
/utf8output	Outputs compiler messages in UTF8 format.
/fullpaths	Instructs the compiler to specify the full path to a file in all messages.
/moduleassemblyname:<string>	Contains the name of the assembly that this module will be a part of.
/main:<classname>	Specifies the class to use for the Main() entry point.

CHAPTER 42

C# Compared to Other Languages

This chapter compares C# to other languages. C#, C++, and Java all share common roots and are more similar to each other than they are to many other languages. Visual Basic isn't as similar to C# as the other languages are, but it still shares many syntactical elements.

Also, this chapter discusses the .NET versions of Visual C++ and Visual Basic, since they're also somewhat different than their predecessors.

Differences Between C# and C/C++

C# code will be familiar to C and C++ programmers, but it has a few big differences and a number of small differences. The following sections give an overview of the differences. For a more detailed perspective, see the MSDN article "C++ -> C#: What You Need to Know to Move from C++ to C#" by Jesse Liberty (`http://msdn.microsoft.com/msdnmag/issues/01/07/ctocsharp/default.aspx`).

A Managed Environment

C# runs in the .NET runtime environment. This not only means many things aren't under the programmer's control but it also means it provides a new set of frameworks. Together, this means a few things are different:

- The garbage collector performs object deletion sometime after the object is no longer used. You can use destructors (a.k.a. *finalizers*) for some cleanup but not in the way you use C++ destructors.
- The C# language doesn't have pointers; well, it has them in `unsafe` mode, but they're rarely used. References are used instead, and they're similar to C++ references without some of the C++ limitations.
- Source is compiled to assemblies, which contain both the compiled code (expressed in the .NET IL) and metadata to describe that compiled code. All .NET languages query the metadata to determine the same information that's contained in C++ `.h` files, and the `include` files are therefore absent.
- Calling native code requires a bit more work.

- No C/C++ runtime library exists. The same things—such as string manipulation, file I/O, and other routines—exist within the .NET Framework libraries and reside in the namespaces that start with `System`.
- C# uses exception handling instead of error returns.

.NET Objects

C# objects all have the ultimate base class `object`, and there's only single inheritance of classes (though there's multiple implementation of interfaces).

You can declare lightweight objects, such as data types, as structs (also known as *value types*), which means they're allocated on the stack instead of the heap.

You can use C# structs and other value types (including the built-in data types) in situations where objects are required by boxing them, which automatically copies the value into a heap-allocated wrapper that's compliant with heap-allocated objects (also known as *reference objects*). This unifies the type system, allowing any variable to be treated as an object but without overhead when unification isn't needed.

C# supports properties and indexers to separate the user model of an object from the implementation of the object, and it supports delegates and events to encapsulate function pointers and callbacks.

C# contains the `params` keyword to provide support similar to `varargs`.

C# Statements

C# statements have high fidelity to C++ statements. A few notable differences exist:

- The `new` keyword means "obtain a new instance of." The object is heap allocated if it's a reference type and stack or inline allocated if it's a value type.
- All statements that test a Boolean condition now require a variable of type `bool`. There's no automatic conversion from `int` to `bool`, so `if (i)` isn't valid.
- `switch` statements disallow fall-through to reduce errors. You can also use `switch` on string values.
- You can use `Foreach` to iterate over objects and collections.
- You can use `Checked` and `unchecked` to control whether arithmetic operations and conversions are checked for overflow.
- Definite assignment requires that objects have a definite value before being used.

Anonymous Methods

C# provides the ability to use anonymous methods, which are a form of inline delegate in which the code that will be executed as part of the delegate is placed inline with the delegate's construction. See Chapter 23 for more details.

C++/CLI doesn't support anonymous methods, but the same functionality is still possible with standard delegates.

Nullable Types

C++/CLI doesn't support nullable types at a language level, but the generic type `System.Nullable<T>`, which ships with the .NET Framework library and is wrapped by C# language nullable types, is available.

Native C++ doesn't have the concept of the value-type/reference-type divide; hence, nullable types have no relevance or meaning in this context.

Iterators

C++/CLI doesn't support iterators, which are a C# language feature that make it easier to write enumerations. Chapter 20 covers iterators. The same functionality is available through enumerators, though it may be considerably more complex to implement in C++/CLI.

Attributes

Attributes are annotations written to convey declarative data from the programmer to other tools or code. That other code might be the runtime environment, a designer, a code-analysis tool, or some other custom tool. You can retrieve attribute information through a process known as *reflection*.

You can write attributes inside square brackets and can place them on classes, members, parameters, and other code elements. Here's an example:

```
[CodeReview("1/1/1999", Comment="Rockin'")]
class Test
{
}
```

Versioning

C# enables better versioning than C++. Because the runtime handles member layout, binary compatibility isn't an issue. The runtime provides side-by-side versions of components if desired and correct semantics when versioning frameworks, and the C# language allows the programmer to specify versioning intent.

Code Organization

C# has no header files; all code is written inline, and although there's preprocessor support for conditional code, there's no support for macros. These restrictions make it both easier and faster for the compiler to parse C# code and also make it easier for development environments to understand C# code.

In addition, C# has no order dependence and no forward declarations. The order of classes in source files is unimportant; you can rearrange classes at will.

Missing C# Features

The following C++ features aren't in C#:

- Multiple inheritance.
- Const member functions or parameters. Const fields are supported.
- Global variables.
- Typedef.
- Conversion by construction.
- Default arguments on function parameters.
- Automatic disposal of stack-declared `IDisposable` objects. C# requires a `using` block. In C++/CLI, the compiler adds the equivalent of the `using` block.
- Automatic generation of a member variable for simple properties. In C++/CLI, you have no need to create a separate property and member variable if all the property accessors do is get and set the member variable. You can use the `property` keyword to instruct the compiler to generate the backing member variable.

Differences Between C# and Java

It's no surprise that there are similarities between C# and Java. You'll see a fair number of differences between them, however. The biggest difference is that C# sits on the .NET Framework and runtime, and Java sits on the Java framework and runtime.

Data Types

C# has more primitive data types than Java. Table 42-1 summarizes the Java types and their C# analogs.

Table 42-1. *Java-to-C# Data Type Mappings*

C# Type	Java Type	Comment
`sbyte`	`byte`	The C# byte is unsigned.
`short`	`short`	
`int`	`int`	
`long`	`long`	
`bool`	`Boolean`	
`float`	`float`	
`double`	`double`	
`char`	`char`	
`string`	`string`	
`object`	`object`	

Table 42-1. *Java-to-C# Data Type Mappings (Continued)*

C# Type	Java Type	Comment
`byte`		Unsigned byte.
`ushort`		Unsigned short.
`uint`		Unsigned int.
`ulong`		Unsigned long.
`decimal`		Financial/monetary type.

In Java, the primitive data types are in a separate world from the object-based types. For primitive types to participate in the object-based world (in a collection, for example), they must be in an instance of a wrapper class, and the wrapper class must be in that collection.

C# approaches this problem differently. In C#, primitive types are stack allocated as in Java, but they're also considered to be derived from the ultimate base class, `object`. This means the primitive types can have member functions defined and called on them. In other words, you can write the following code:

```
using System;
class Test
{
    public static void Main()
    {
        Console.WriteLine(5.ToString());
    }
}
```

The constant `5` is of type `int`, and the `ToString()` member is defined for the `int` type, so the compiler can generate a call to it and pass the `int` to the member function as if it's an object.

This works well when the compiler knows it's dealing with a primitive, but it doesn't work when a primitive needs to work with heap-allocated objects in a collection. Whenever a primitive type is used in a situation where a parameter of type `object` is required, the compiler will automatically box the primitive type into a heap-allocated wrapper. Here's an example of boxing:

```
using System;
class Test
{
    public static void Main()
    {
        int v = 55;
        object o = v;        // box v into o
        Console.WriteLine("Value is: {0}", o);
        int v2 = (int) o;    // unbox back to an int
    }
}
```

In this code, the integer is boxed into an `object` and then passed off to the `Console.WriteLine()` member function as an object parameter. We declared the object variable

for illustration only; in real code, v would be passed directly, and the boxing would happen at the call site. You can extract the boxed integer by a `cast` operation, which will extract the boxed `int`.

Extending the Type System

The primitive C# types (with the exception of `string` and `object`) are also known as *value types*, because variables of those types contain actual values. Other types are known as *reference types*, because those variables contain references.

In C#, a programmer can extend the type system by implementing a custom value type. These types are implemented using the `struct` keyword and behave similarly to built-in value types; they're stack allocated, can have member functions defined on them, and are boxed and unboxed as necessary. In fact, the C# primitive types are all implemented as value types, and the only syntactical difference between the built-in types and user-defined types is that you can write the built-in types as constants.

To make user-defined types behave naturally, C# structs can overload arithmetic operators so that numeric operations can be performed and can overload conversions so implicit and explicit conversions can be performed between structs and other types. C# also supports overloading on classes.

You write a `struct` using the same syntax as a `class`, except that a `struct` can't have a base class (other than the implicit base class `object`), though it can implement interfaces.

Classes

C# classes are quite similar to Java classes, with a few important differences relating to constants, base classes and constructors, static constructors, virtual functions, hiding, and versioning, accessibility of members, `ref` and `out` parameters, and identifying types.

Constants

Java uses `static final` to declare a class constant. C# replaces this with `const`. In addition, C# adds the `readonly` keyword, which is used in situations where the constant value can't be determined at compile time. `Readonly` fields can be set only through an initializer or a class constructor.

Base Classes and Constructors

C# uses the C++ syntax both for defining the base class and interfaces of a class and for calling other constructors. A C# class that does this might look like this:

```
public class MyObject: Control, IFormattable
{
    public MyObject(int value)
    {
        this.value = value;
    }
        public MyObject()  : base(value)
    {
    }
    int value;
}
```

Static Constructors

As well as using a static initialization block, C# provides static constructors, which are written using the `static` keyword in front of a parameterless constructor.

Virtual Functions, Hiding, and Versioning

In C#, all methods are nonvirtual by default, and you must specify `virtual` explicitly to make a function virtual. Because of this, C# doesn't contain any final methods, but you can achieve the equivalent of a final class using `sealed`.

C# provides better versioning support than Java, and this results in a few small changes. Because versioning is specified explicitly in C#, adding a virtual function in a base class won't change a program's behavior. Consider the following:

```
public class B
{
}
public class D: B
{
    public void Process(object o) {}
}
class Test
{
    public static void Main()
    {
        D d = new D();
        d.Process(15);    // make call
    }
}
```

If the provider of the base class adds a process function that's a better match, the behavior will change:

```
public class B
{
    public void Process(int v) {}
}
public class D: B
{
    public void Process(object o) {}
}
class Test
{
    public static void Main()
    {
        D d = new D();
        d.Process(15);    // make call
    }
}
```

In Java, this will now call the base class's implementation, which is unlikely to be correct. In C#, the program will continue to work as before.

To handle the similar case for virtual functions, C# requires that you specify the versioning semantics explicitly. If `Process()` had been a virtual function in the derived class, Java would assume that any base class function that matched in signature would be a base for that virtual, which is unlikely to be correct.

In C#, virtual functions are overridden only if the `override` keyword is specified. See Chapter 11 for more information.

Accessibility of Members

In addition to `public`, `private`, and `protected` accessibility, C# adds `internal`. You can access members with `internal` accessibility from other classes within the same project but not from outside the project.

Operator Overloading

C# allows the user to overload most operators for both classes and structs so that objects can participate in mathematic expressions. C# doesn't allow the overloading of more complex operators such as member access, function invocation, assignment, or the `new` operator, because overloading these can make code much more complicated.

ref and out Parameters

In Java, parameters are always passed by value. C# allows parameters to be passed by reference by using the `ref` keyword. This allows the member function to change the value of the parameter.

C# also allows parameters to be defined using the `out` keyword, which functions the same as `ref`, except that the variable passed as the parameter doesn't have to have a known value before the call.

Enumerations

The C# enum type is akin to the C++ enum type and is used similarly. Implicit conversions between an enum and its underlying type are more restricted, however.

Identifying Types

Java uses the `GetClass()` method to return a `Class` object, which contains information about the object on which it's called. The `Type` object is the .NET analog to the `Class` object; you can obtain it in several ways:

- By calling the `GetType()` method on an instance of an object
- By using the `typeof` operator on the name of a type
- By looking up the type by name using the classes in `System.Reflection`

Interfaces

Although Java interfaces can have constants, C# interfaces cannot. When implementing interfaces, C# provides explicit interface implementation. This allows a class to implement two interfaces from two different sources that have the same member name, and you can also use it to hide interface implementations from the user. For more information, see Chapter 10.

Properties and Indexers

Java programs often use the property idiom by declaring `get` and `set` methods. In C#, a property appears to the user of a class as a field but has a get and set accessor to perform the read and/or write operations.

An indexer is similar to a property, but instead of looking like a field, an indexer appears as an array to the user. Like properties, indexers have get and set accessors, but unlike properties, an indexer can be overloaded on different types. This enables a database row that can be indexed both by column number and by column name and a hash table that can be indexed by hash key.

Delegates and Events

When an object needs to receive a callback in Java, an interface specifies how the object must be formed, and a method in that interface is called for the callback. You can use a similar approach in C# with interfaces.

C# adds delegates, which can be thought of as type-safe function pointers. A class can create a delegate on a function in the class, and then that delegate can be passed off to a function that accepts the delegate. That function can then call the delegate.

C# builds upon delegates with events, which are used by the .NET Framework. Events implement the publish-and-subscribe idiom; if an object (such as a control) supports a click event, any number of other classes can register a delegate to be called when that event fires.

Attributes

Attributes are annotations written to convey declarative data from the programmer to other code. That other code might be the runtime environment, a designer, a code-analysis tool, or some other custom tool. You can retrieve attribute information through a process known as *reflection*.

You write attributes inside square brackets, and you can place them on classes, members, parameters, and other code elements. Here's an example:

```
[CodeReview("1/1/1999", Comment="Rockin'")]
class Test
{
}
```

Statements

Statements in C# will be familiar to the Java programmer, but there are a few new statements and a few differences in existing statements to keep in mind.

import vs. using

In Java, the `import` statement locates a package and imports the types into the current file.

In C#, this operation is split. You must explicitly specify the assemblies that a section of code relies upon, either on the command line using `/r` or in the Visual Studio IDE. The most basic system functions (currently those contained in `mscorlib.dll`) are the only ones imported automatically by the compiler.

Once an assembly has been referenced, the types in it are available for use, but you must specify them using their fully qualified names. For example, the regular expression class is named `System.Text.RegularExpressions.Regex`. You could use that class name directly, or you could use a `using` statement to import the types in a namespace to the top-level namespace. With the following `using` clause, you can specify the class merely by using `Regex`:

```
using System.Text.RegularExpressions;
```

Also, a variant of the `using` statement allows aliases for types to be specified if there's a name collision.

Overflows

Java doesn't detect overflow in conversions or mathematical expressions.

In C#, you can control the detection of these by the `checked` and `unchecked` statements and operators. Conversions and mathematical operations that occur in a `checked` context will throw exceptions if the operations generate overflow or other errors; such operations in an `unchecked` context will never throw errors. The `/checked` compiler flag controls the default context.

Unsafe Code

Unsafe code in C# allows the use of pointer variables, and you use it when performance is extremely important or when interfacing with existing software, such as COM objects or native C code in DLLs. The `fixed` statement is used to "pin" an object so that it won't move if a garbage collection occurs.

Because unsafe code can't be verified to be safe by the runtime, it can be executed only if it's fully trusted by the runtime. This prevents execution in download scenarios.

Strings

You can index the C# `string` object to access specific characters. Comparison between strings evaluates the values of the strings rather than the references to the strings.

String literals are also a bit different; C# supports escape characters within strings that are used to insert special characters. The string `\t` will be translated to a tab character, for example.

Generics

Java generics and C# generics are very different. As discussed in Chapter 17, generics in C# have first-class runtime support, and a C# generic type is fully compatible with a generic type produced in any other .NET language. In contrast, Java generics are entirely a compile-time concept, and the compiler will simply replace the type parameter with an object reference in a process referred to as *type eraser*. This means the Java bytecode (which is Java's equivalent to MSIL) is entirely nongeneric, minimizing the upgrade burden on Java Virtual Machine (JVM) implementers.

The cost of implementing generics at compile rather than runtime is that the efficiency gains of strong runtime typing aren't available in Java, and boxing/unboxing will still occur in Java for generic collections of primitive types. In addition, by implementing generics as a language rather than platform features, it isn't possible to use or query for generic information at runtime using technologies such as reflection.

Documentation

The XML documentation in C# is similar to Javadoc, but C# doesn't dictate the organization of the documentation, and the compiler checks for correctness and generates unique identifiers for links.

Miscellaneous Differences

A few miscellaneous differences exist:

- The >>> operator isn't present, because the >> operator has different behavior for signed and unsigned types.
- The `is` operator is used instead of `instanceof`.
- There's no labeled break statement; `goto` replaces it.
- The `switch` statement prohibits fall-through, and you can use `switch` on string variables.
- There's only one array declaration syntax: `int[] arr`.
- C# allows a variable number of parameters using the `params` keyword.

Differences Between C# and Visual Basic 6

C# and Visual Basic 6 are fairly different languages. C# is an object-oriented language, and VB 6 has only limited object-oriented features. VB .NET adds object-oriented features to the VB language, and it may therefore be instructive to also study the VB .NET documentation.

Code Appearance

In VB, statement blocks end with some sort of `END` statement, and multiple statements can't appear on a single line. In C#, blocks are denoted using braces ({}), and the location of line breaks doesn't matter, because a semicolon indicates the end of a statement. Though it might be bad form and ugly to read, you can write the following in C#:

```
for (int j = 0; j < 10; j++) {if (j == 5) Func(j); else return;}
```

That line means the same as this:

```
for (int j = 0; j < 10; j++)
{
    if (j == 5)
        Func(j);
    else
        return;
}
```

This constrains the programmer less, but it also makes agreements about style more important.

Data Types and Variables

Although there's a considerable amount of overlap in data types between VB and C#, there are some important differences; therefore, a similar name may mean a different data type.

The most important difference is that C# is stricter on variable declaration and usage. All variables must be declared before they're used, and they must be declared with a specific type—there's no `Variant` type that can hold any type.[1]

You can make variable declarations simply by using the name of the type before the variable; no `dim` statement exists.

Conversions

Conversions between types are also stricter than in VB. C# has two types of conversions: implicit and explicit. *Implicit* conversions are those that can't lose data—that's where the source value will always fit into the destination variable. For example:

```
int    v = 55;
long x = v;
```

Assigning `v` to `x` is allowed because `int` variables can always fit into `long` variables.

Explicit conversions, on the other hand, are conversions that can lose data or fail. Because of this, the conversion must be explicitly stated using a cast:

```
long    x = 55;
int v = (int) x;
```

Though in this case the conversion is safe, the `long` can hold numbers that are too big to fit in an `int`, and therefore the cast is required.

If detecting overflow in conversions is important, you can use the `checked` statement to turn on the detection of overflow. See Chapter 15 for more information.

1. The object type can contain any type, but it knows exactly what type it contains.

Data Type Differences

In Visual Basic, the integer data types are `Integer` and `Long`. In C#, these are replaced with the types `short` and `int`. There's a `long` type as well, but it's a 64-bit (8-byte) type. This is something to keep in mind, because if `long` is used in C# where `Long` would have been used in VB, programs will be a bit bigger and a bit slower. `Byte`, however, is merely renamed to `byte`.

C# also has the unsigned data types `ushort`, `uint`, and `ulong` and the signed byte `sbyte`. These are useful in some situations, but not all languages in .NET can use them, so you should use them only as necessary.

The floating-point types `Single` and `Double` are renamed `float` and `double`, and the `Boolean` type is known simply as `bool`.

Strings

Many of the built-in functions that are present in VB don't exist for the C# string type. Functions exist to search strings, extract substrings, and perform other operations; see the documentation for the `System.String` type for details.

String concatenation takes place using the + operator rather than the & operator.

Arrays

In C#, the first element of an array is always index 0; also, there's no way to set upper or lower bounds and no way to `redim` an array. However, an `ArrayList` in the `System.Collection` namespace does allow resizing. The `System.Collection` namespace also contains many other useful collection classes.

Operators and Expressions

The operators that C# uses have a few differences from VB, and the expressions will therefore take some getting used to. Table 42-2 shows the C# equivalent for some common VB operators.

Table 42-2. *C# Equivalent for VB Operators*

VB Operator	C# Equivalent
`^`	None. See `Math.Pow()`.
`Mod`	The equivalent is `%`.
`&`	The equivalent is `+`.
`=`	The equivalent is `==`.
`<>`	The equivalent is `!=`.
`Like`	None. `System.Text.RegularExpressions.Regex` does some of this, but it's more complex.
`Is`	None. The C# `is` operator means something different.
`And`	The equivalent is `&&`.
`Or`	The equivalent is `||`.

Table 42-2. *C# Equivalent for VB Operators (Continued)*

VB Operator	C# Equivalent
`Xor`	The equivalent is `^`.
`Eqv`	None. `A Eqv B` is the same as `!(A ^ B)`.
`Imp`	None.

Classes, Types, Functions, and Interfaces

Because C# is an object-oriented language,[2] the class is the major organizational unit; rather than having code or variables live in a global area, they're always associated with a specific class. This results in code that's structured and organized quite differently than VB code, but still some common elements exist. You can still use properties, but they have a different syntax and no default properties.

Functions

In C#, function parameters must have a declared type, and you can use `ref` instead of `ByRef` to indicate that the value of a passed variable may be modified. You can achieve the equivalent of the `ParamArray` function by using the `params` keyword.

Control and Program Flow

C# and VB have similar control structures, but the syntax used is a bit different.

If Then

C# has no `Then` statement; after the condition comes the statement or statement block that should be executed if the condition is true, and after that statement or block comes an optional `else` statement.

For example, you can rewrite the following VB code:

```
If size < 60 Then
    value = 50
Else
    value = 55
    order = 12
End If
```

as follows:

```
if (size < 60)
    value = 50;
```

2. See Chapter 1 for more information.

```
else
{
    value = 55;
    order = 12;
}
```

C# has no `ElseIf` statement in C#, but it has a simple statement, `Else If`.

For

The syntax for `for` loops is different in C#, but the concept is the same, except that in C# the operation performed at the end of each loop must be explicitly specified. In other words, you can rewrite the following VB code:

```
For i = 1 To 100
    ' other code here
Next
```

as follows:

```
for (int i = 0; i < 10; i++)
{
    // other code here
}
```

For Each

C# supports the `For Each` syntax through the `foreach` statement, which can be used on arrays, collections classes, and other classes that expose the proper interface.

Do Loop

C# has two looping constructs to replace the `Do Loop` construct. The `while` statement loops while a condition is true, and `do while` works the same way, except that one trip through the loop is ensured even if the condition is false. For example, you can rewrite the following VB code:

```
I = 1
fact = 1
Do While I <= n
    fact = fact * I
    I = I + 1
Loop
```

as follows:

```
int I = 1;
int fact = 1;
while (I <= n)
{
    fact = fact * I;
    I++;
}
```

A loop can be exited using the `break` statement or continued on the next iteration using the `continue` statement.

Select Case

The `switch` statement in C# does the same thing as `Select Case`. You can rewrite this VB code:

```
Select Case x
    Case 1
        Func1
    Case 2
        Func2
    Case 3
        Func2
    Case Else
        Func3
End Select
```

as follows:

```
switch (x)
{
    case 1:
        Func1();
        break;
    case 2:
    case 3:
        Func2();
        break;
    default:
        Func3();
        break;
}
```

On Error

C# doesn't have any `On Error` statement. Error conditions in .NET are communicated through exceptions. See Chapter 4 for more details.

Missing Statements

C# doesn't have `With`, `Choose`, or the equivalent of `Switch`. It also doesn't have any `CallByName` feature, but you can do this through reflection.

Other .NET Languages

Visual C++ and Visual Basic have both been extended to work in the .NET world.

In the Visual C++ world, a set of Managed Extensions have been added to the language to allow programmers to produce and consume components for the CLR. The Visual C++ model allows the programmer more control than the C# model; the user is allowed to write both managed (garbage-collected) and unmanaged (using `new` and `delete`) objects.

You can create a .NET component using keywords to modify the meaning of existing C++ constructs. For example, when you place the `__gc` keyword in front of a class definition, you enable the creation of a managed class and restrict the class from using constructs that can't be expressed in the .NET world (such as multiple inheritance). You can also use the .NET system classes from the Managed Extensions.

The Managed Extensions for C++ that existed in Visual C++ 2002 and 2003 have been superceded by a new binding of C++ to .NET called C++/CLI that will be part of Visual C++ 2005. C++/CLI offer a cleaner and more consistent way to use .NET features and also allow C++ code to be written and compiled so that it can be verifiably type-safe. For developers who left C++ because of the ugliness and difficulty of Managed C++, it's worth evaluating C++/CLI.

In addition to the Microsoft languages, several third-party languages have been announced for the .NET platform. See `http://www.gotdotnet.com` for more information.

CHAPTER 43

C# Resources and the Future

This chapter provides some resources for learning more about C# and provides some ideas about how C# will evolve in the future.

C# Resources

Several C# resources have appeared on the Web. The following sections list some of them.

MSDN

MSDN (`http://msdn.microsoft.com/netframework`) is the main Microsoft site for all things .NET. It has news, articles, columns, and sample code. The C# subsite is at `http://msdn.microsoft.com/vcsharp/` and contains links to the blogs of many of the C# team and C# MVPs (including links to the blogs of this book's authors). The C# subsite also includes the full product and language reference documentation.

The .NET 2.0 Framework site is currently at `http://msdn2.microsoft.com`.

GotDotNet

GotDotNet (`http://www.gotdotnet.com`) is a Microsoft-operated community site. It has some of the same content as the MSDN site, but it also has a user-contribution area.

C-Sharp Corner

C-Sharp Corner (`http://www.c-sharpcorner.com`) is a site dedicated only to C#. It's somewhat like GotDotNet, but it deals only with C#, not the whole .NET universe.

CodeGuru

CodeGuru (`http://www.codeguru.com`) is an EarthWeb site (the EarthWeb family includes Internet.com and Developer.com), and it includes many sections dedicated to .NET language and technologies.

The Code Project

The Code Project (`http://www.codeproject.com`) is a community-based Web site that hosts a huge range of .NET content, the bulk of which has been submitted by the Web site's members. The site currently has more than 8,000 "free C++, C#, and .NET articles; code snippets; discussions; news; and the best bunch of developers on the Net." The quality of the Code Project articles is quite variable, and an online rating system is available to provide readers with an indication of the quality of a particular contribution.

PInvoke.NET

PInvoke.NET (`http://www.pinvoke.net`) is a wiki-style site that provides a huge range of PInvoke signatures for calling native functions on the Windows platform. A wiki Web site allows users to edit and add to the content, which means the collection of method signatures is constantly being added to and enhanced.

DotNet Books

If there's an existing, new, or upcoming book about C# or other .NET topics, it will probably be listed at DotNet Books (`http://www.dotnetbooks.com`).

The Future of C#

The C# compiler specification was submitted for standardization to the European Computer Manufacturers Association (ECMA). The standardization process is underway in Technical Committee 39, the same group that standardized ECMAScript (often known as JavaScript or JScript). In addition to the C# language, a subset of the CLR, known as the CLI, is undergoing standardization in the same committee. The C# and CLI standards were ratified in April 2003.

The goal is to standardize enough of the language and runtime so that useful programs can be written, roughly analogous to what's available with C++ and the C++ runtime library. Current specifications from this process are available at `http://msdn.microsoft.com/net/ecma`.

C# 2.0 is currently undergoing the same standardization process that C# 1.0 underwent. One of the most exciting advances in this area is that generics have now been added to the CLI spec, meaning they will eventually be available in a wide range of languages. You can view the current specifications at `http://msdn.microsoft.com/vcsharp/team/language`.

INDEX

D

E

F

G

H

I

J

L

M

N

R

S

T

U

V

W

X

Y