BOOKS FOR PROFESSIONALS BY PROFESSIONALS™

Foreword by Anders Hejlsberg, architect of C#, Delphi, and Turbo Pascal

Eric Gunnerson

A Programmer's Introduction to C#

Learn C# from a member of Microsoft's C# design team

Teaches you to program with C# and to author .NET components

Get up-to-speed quickly on the inner workings of C#

Apress™

A Programmer's Introduction to C#

by Eric Gunnerson

ISBN: 1893115860

Apress © 2000, 358 pages

This book takes the C programmer through the all the details—from basic to advanced-- of the new Microsoft C# language.

Companion Web Site

Table of Contents

Colleague Comments

Back Cover

Synopsis

Written as an introduction to the new C#, this guide takes the experienced C programmer a few steps beyond the basics. It covers objects, data types, and flow control, and even delves into some background on the new Microsoft NET Frameworks environment. Keeping in mind that this is for those familiar with C (and even Java), the book goes into some of the advanced features and improvements found in this new language. It also offers a comparison between C#, C++, Visual Basic, and Java.

Table of Contents

Back Cover

- Provides in-depth information about the functionality of the language and C# "Quick Start"
- Shows you how to write components that fit seamlessly into the .NET Frameworks
- Includes C# reference information tailored for C++, Java and Visual Basic Programmers
- Suitable for intermediate to advanced developers and includes coverage of advanced topics in C#

Eric Gunnerson, A member of the Microsoft C# design team, has written a comprehensive C# tutorial addressed to the experienced programmer. *A Programmer's Introduction to C#* explains how C# works, why it was designed the way it was, and how C# fits into Microsoft's new .NET Frameworks. This book teaches programmers how to write C# components and how to truly leverage the power of the new .NET Runtime.

Gunnerson's first chapters are for the impatient programmer. In them, he provides an introduction to object-oriented programming with C# along with a C# "Quick Start" for those who want a fast track to programming in C#. This is followed by a more comprehensive section in which he uses his unique insider's view to explain each of the new C# language features in detail. He covers fundamentals such as classes, structs, attributes, statements and flow of execution, arrays, delegates and events, exception handling, and the unique interoperability provided by the .NET Frameworks.

In the final portion of the book, Gunnerson provides a useful overview of the .NET Frameworks. A section on the .NET Common Language Runtime and Framworks shows how to write components that function well in the runtime and how to use the basic runtime features (such as I/O). Gunnerson also devoted time to more advanced topics such as regular expressions and collections. Final chapters include Making Friends with the .NET Frameworks, System.Array and the Collection Classes, .NET Fraeworks Overview, Deeper into C# and Defensive Programming. Also included is a detailed C# language comparison that will be indispensable for programmers currently working in C++, Java, or Visual Basic.

All of the source code for this book in online at http://www.apress.com.

About the Author

Eric Gunnerson is a software design engineer in Microsoft's Visual C++ QA group and a member of the C# design team. In the course of his professional career, he has worked primarily on database products and tools – and is proud of the fact that nearly half of the companies he has worked for remain in business.

A Programmer's Introduction to C#

ERIC GUNNERSON

 ISBN (pbk): 1-893115-86-0

Printed and bound in the United States of America 2345678910

Editorial Directors: Dan Appleman, Gary Cornell, Karen Watterson
Technical Reviewers: David Staheli, Shawn Vita, Gus Perez, Jerry Higgins, Brenton Webster
Editor: Andy Carroll
Projects Manager: Grace Wong
Production Editor: Janet Vail
Page Compositor and Soap Bubble Artist: Susan Glinert
Artist: Karl Miyajima
Indexer: Nancy Guenther
Cover and Interior Design: Derek Yee Design

Distributed to the book trade in the United States by Springer-Verlag New York, Inc., 175 Fifth Avenue, New York, NY, 10010 and outside the United States by Springer-Verlag GmbH & Co. KG, Tiergartenstr. 17, 69112 Heidelberg, Germany
In the United States, phone 1-800-SPRINGER orders@springer-ny.com ; http://www.springer-ny.com Outside the United States, contact orders@springer.de; http://www.springer.de ; fax +49 6221 345229
For information on translations, please contact Apress directly at 901 Grayson Street, Suite 204, Berkeley, CA, 94710 Phone: 510-549-5931; Fax: 510-549-5939; info@apress.com ; http://www.apress.com

Dedication

To Tony Jongejan, for introducing me to programming and being ahead of his time.

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 would 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 for my cat for holding my arms down while I was writing.

Foreword

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 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 do not require heap allocation.
- *To enable 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 is 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 This Book

C# IS ONE OF THE MOST EXCITING projects I've ever had the privilege to work on. There are many languages with different strengths and weaknesses, but once in a while a new language comes along that that meshes well with the hardware, software, and programming approaches of a specific time. I believe C# is such a language. Of course, language choice is often a "religious issue." [1]

I've structured this book as a tour through the language, since I 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, there's Chapter 3, "C# QuickStart," which is a quick overview of the language, and gives enough information to start coding.
Chapter 34, "C# Compared to Other Languages," offers language-specific comparisons for C++, VB, and Java for programmers attuned to a specific language, or for those who like to read comparisons.

As I write this, it's early August 2000, and the Visual Studio version that will contain C# has yet to reach beta. The language syntax is fairly stable, but there will undoubtedly be some items changed "around the edges." See Chapter 35, "C# Futures," for some information on what is in store for the future versions.
If you have comments about the book, you can reach me at gunnerso@halcyon.com. All source code can be downloaded from the Apress Web site at http://www.apress.com.

[1] See the Jargon File (//www.jargonfile.org) for a good definition of "religious issue."

Introduction

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 it's own strengths and weaknesses. Though I *could* use my router to trim a board to length, it would be much easier if I used a miter saw. Similarly, I could use a language like LISP to write a graphics-intensive game, but it would probably be easier to use C++.
C# (pronounced "C sharp") is the native language for the .NET Common Language Runtime. It has been designed to fit seamlessly into the .NET Common Language Runtime. 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 Common Language Runtime is central to many things in C#, Chapter 2, "The .NET Runtime Environment," will introduce the important parts of it—at least, those that are important to the C# language.

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 Common Language Runtime, it provides a language to use for component-oriented soft- ware, 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 Common Language Runtime 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 Frameworks, 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 is robust and durable is very 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,[1] 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 is simple, safe, and straightforward. Error handling is not 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.

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.[2] The .NET Common Language Runtime 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. [3]
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 very 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 Common Language Runtime, which provides extensive library support for general programming tasks and application-specific tasks. The .NET Runtime, Frameworks, and languages are all tied together by the Visual Studio environment, providing one-stop-shopping for the .NET programmer.

[1] It's not that C++ memory management is conceptually hard; it isn't in most cases, though 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.

[2] Usually. There are details that sometimes make this a bit tougher in practice.

[3] For C++ code, Visual C++ has been extended with "Managed Extensions" that make it possible to create .NET components. More information on these extensions can be found on the Microsoft web site.

The C# Compiler and Other Resources

THERE ARE 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 the Visual Studio.NET. The beta of Visual Studio.NET will be available in the fall of 2000.

To find out more about getting the .NET SDK or the Visual Studio.NET 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, the system.winforms.dll file must be specified 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.

There are numerous Web sites devoted to .NET information. Links to these resources also can be found at the Apress Web site.

Chapter 1: Object-Oriented Basics

Overview

THIS CHAPTER IS AN INTRODUCTION to object-oriented programming. Those who are familiar with object-oriented programming will probably want to skip this section.

There are 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 Is 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 is composed of 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, address), work information (job title, 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 is 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.

First, inheritance is an example of what is 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 importantly, 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 only be used when the advantages that it brings are needed[2]. See the coming section on “Polymorphism and Virtual Functions.”
In the .NET Common Language Runtime, all objects are inherited from the ultimate base class named `object`, and there is only single inheritance of objects (i.e., an object can only be derived from one base class). This does prevent 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 tradeoff. The .NET Runtime does allow multiple inheritance in the form of interfaces, which cannot contain implementation. Interfaces will be discussed in Chapter 10, "Interfaces."

Containment

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

The answer is containment, also known as aggregation. Rather than saying that 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 array, or hash table).

The default design choice should be containment, and you should switch to inheritance only if needed (i.e., if there really is an “is-a” relationship).

[1] At this point there should perhaps be an appropriate comment about standing “on the shoulders of giants…”

[2]Perhaps there should be a paper called “Multiple inheritance considered harmful.” There probably is one, someplace.

Polymorphism and Virtual Functions

A while back I was writing a music system, and I decided that I wanted to be able to support both WinAmp and Windows Media Player as playback engines, but I didn’t want all of my code to have to know which engine it was using. I 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.
In this case, the abstract class was called `MusicServer`, and it had functions like `Play()`, `NextSong()`, `Pause()`, etc. Each of these functions was declared as abstract, so that 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 is 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 that 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 is 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, etc.

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

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. The most important reasons for doing this are these:

- The user can't change private things in the object, which reduces the chance that 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 is 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 only be accessed from within the class; public fields can be accessed through any instance of the class. Having more public fields often makes debugging much tougher.

This subject will be explored further in Chapter 5, "Classes 101."

Chapter 2: The .Net Runtime Environment

Overview

IN THE PAST, WRITING MODULES that could be called from multiple languages was difficult. Code that is written in Visual Basic can't be called from Visual C++. Code that is 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 very specific behavior, and Visual Basic uses its own execution engine, also with its own specific—and different—behavior.

And so COM was created, and it's 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 got used extensively when writing COM components, and 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 of working together.

Further, the world was tough for library providers, as there was no one choice that would work in all markets. If the writer thought the library was targeted toward the Visual Basic crowd, it would 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 would ignore the Visual Basic programmers.

Sometimes a library would be written for both types of users, but this usually meant there were some compromises. To send email on a Windows system, there is a choice between Collaboration Data Objects (CDO), a COM-based interface that can be called from both languages but doesn't do everything,[1] and native MAPI functions (in both C and C++ versions) that can access all functions.

The .NET Runtime is designed to remedy this situation. There is one way of describing code (metadata), and one runtime and library (the Common Language Runtime and Frameworks). The following diagram shows how the .NET Runtime is arranged:

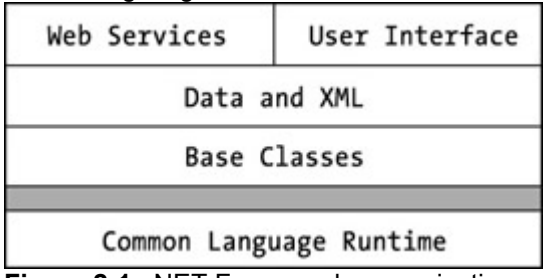


Figure 2-1. .NET Frameworks organization

The Common Language Runtime 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 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.
To understand how C# works, it is important to understand a bit about the .NET Runtime and Frameworks. The following section provides an overview; and more detailed information can be found later in the book in the Chapter 31, "Deeper into C#."

[1]Presumably this is because it is difficult to translate the low-level internal design into something that can be called from an automation interface.

[2]A way to expose a programmatic interface via a web server.

The Execution Environment

This section was once titled, "The Execution Engine," but .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 the library providers.

The environment also supports the use of existing code in C# code, either through calling functions in 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. There is no need to switch between error codes, HRESULTs, and exceptions.

Finally, the environment contains the Base Class Libraries (BCL), which provide the functions traditionally found in runtime libraries, plus a few new ones. Some of the functionality the BCL provides includes:

- Collection classes, such as queues, arrays, stacks, and hash tables
- Database access classes
- IO classes
- WinForms classes, for creating user interfaces
- Network classes

Outside the base class runtime, there are many other components that handle UI and perform other sophisticated operations.

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 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 is 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, Visual C++ with Managed Extensions, C#, and JScript. 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 only runs with the components it shipped with, rather than with different versions shipped by other applications.

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

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 is part of the .NET world, the metadata for that object records all the information that is 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 very nice for the both the producer and 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 makes use of 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 is not strongly typed, so most software will choose to reference the metadata at compile time rather than runtime, but it is a very 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 are not known until runtime.

Assemblies

In the past, a finished software package might have been released as an executable, DLL and LIB files, a DLL containing a COM object and a typelib, or 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 is converted to an intermediate form known as "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 that are 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; there is enough information 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. Further, 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 additional expressibility, so code that is 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. More information on the CLS can be found in .NET SDK, in the section on "Cross-Language Interoperability."
A user writing C# code can indicate that it is supposed to be CLS compliant, and the compiler will flag any non-compliant areas. For more information on the specific restrictions placed on C# code by CLS compliance, see the "CLS Compliance" section in Chapter 31, "Deeper into C#."

Attributes

To transform a class into a component, some additional information is often required, 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. [4]
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 at a later time. Attributes provide a general mechanism for doing this, and they are 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] Anybody who has ever tried to do COM programming without a typelib should understand the problem with this.

Chapter 3: C# Quickstart

Overview

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.

Hello, Universe

As a supporter of SETI,[1] I thought that it would 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. 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 there are no global functions in C#, 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.

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

Namespaces and Using

Namespaces in the .NET Runtime are used to 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.

Namespaces can also be thought of as 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` allows the user to omit namespaces when using a type, so that the types can be more easily referenced.
`Using` is merely a shortcut that reduces the amount of typing that is required when referring to elements, as the following table indicates:

USING CLAUSE	SOURCE LINE
<none>	System.Console.WriteLine("Hello");
`using` System	Console.WriteLine("Hello");

Note that `using` cannot be used with a class name, so that the class name could be omitted. In other words, `using System.Console` is not allowed.
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 very long name if the class is deeply nested, so there 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 they should be used in most real code.

Namespaces and Assemblies

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

To reference objects located in other assemblies, the name of the assembly file must be passed to the compiler. This can be done on the command line using the `/r:<assembly>` option, or from within the Visual Studio IDE by adding a reference to the C# project.
Typically, there is a correlation between the namespace that 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.

The exact name of the assembly that an object is contained in can be found in the documentation for that object.

Basic Data Types

C# supports the usual set of data types. For each data type that C# supports, there is a corresponding underlying .NET Common Language Runtime type. For example, the `int` type in C# maps to the `System.Int32` type in the runtime. `System.Int32` could be used in most of the places where `int` is used, but that isn't recommended because it makes the code tougher to read.
The basic types are described in the following table. The runtime types can all be found in the `System` namespace of the .NET Common Language Runtime.

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`		`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 is 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. The wrapper is marked so that the system knows that it contains an int. 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`. That allows the following to be written:

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

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 either the multidimensional or jagged forms. More advanced data structures, such as stacks and hash tables, can be found in the `System.Collections` namespace.

Classes, Structs, and Interfaces

In C#, the `class` keyword is used to declare a reference (heap allocated) type, and the `struct` keyword is used to declare 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. The following diagram details how these work:

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

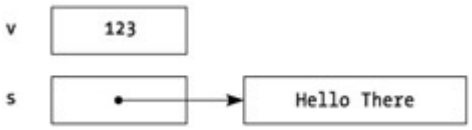


Figure 3-1. Value and reference type allocation

C# and the .NET Runtime do not 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. The `foreach` statement is used to iterate over arrays and collections, the `lock` statement is used for mutual exclusion in threading scenarios, and the `checked` and `unchecked` statements are used to control overflow checking in arithmetic operations and conversions.

Enums

Enumerators are used to 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
}
```

Enumerators are covered in more detail in Chapter 20, "Enumerators."

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 is using it. They are used most heavily as the basis for events, which allow a delegate to easily be registered for an event. They are discussed in Chapter 22, “Delegates.”

Delegates and events are heavily used by the .NET Frameworks.

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 X
    {
        get
        {
            return(x);
        }
        set
        {
            x = value;
            // draw the object here.
        }
    }
    int x;
}
class Test
{
    public static void Main()
    {
        Circle c = new Circle();
        c.X = 35;
    }
}
```

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

Attributes

Attributes are used in C# and the .NET Frameworks to communicate declarative information from the writer of the code to other code that is interested in the information. This could be used 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.

Chapter 4: Exception Handling

Overview

IN MANY PROGRAMMING BOOKS, exception handling warrants a chapter somewhat late in the book. In this book, however, it's very 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 very 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 that classes must be used without really introducing them. Read the following section for flavor; classes will be covered in detail in the next chapter.

What's Wrong with Return Codes?

Most programmers have probably written code that looked 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 above was written as

```
CallFunction();
```

any error return would be thrown away. That's where bugs come from.
There are many different models 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 is contained within the exception object.

The Exception Hierarchy

All C# exceptions derive from the class named `Exception`, which is part of the Common Language Runtime[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. Returning to the example:

```
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 is executed.

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();
       }
       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.

[1] This is true of .NET classes in general, but there are some cases where this might not hold true.

Passing Exceptions on to the Caller

It's sometimes the case that there's not much that can be done when an exception occurs; it really has to be handled by the calling function. There are 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 e;
        }
    }
}
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 while 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.

```
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 is 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.

```
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: Exception
{
    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.

Finally

Sometimes, when writing a function, there will be some cleanup that needs to be done 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 = Int32.FromString(line);
              line = t.ReadLine();
              sum = Int32.FromString(line);
              CalculateAverage(count, sum);
          }
          f.Close();
      }
          // 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 `DivideByZero-Exception`, 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 there is an exception or not. 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 that they are properly destroyed if an exception is thrown. The required tracking code both adds 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 very inexpensive to implement and imposes little runtime overhead on the program when the exceptional case doesn't occur.

Design Guidelines

Exceptions should be used 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, there should be no exceptions thrown.

Conversely, don't use return values to communicate information that would be better contained in an exception.
If there's a good predefined exception in the `System` namespace that 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

Overview

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# that will be used often, and the later sections describe things that won't be used as often, depending on what kind of code is being written.

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, zeroes out all data members of the class, and returns a reference to the instance. A reference is simply a way to refer to an instance. [1]

There is 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]

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);
   }
}
```

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

In addition to the data members, there is a constructor for the class, which is a special function that is 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, the `this.<name>` syntax must be used.
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, there is a `Test` class that contains a `Main` function that is 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 out the values of `x` and `y`.

[1]For those of you used to pointers, a reference is pointer that you can only assign to and dereference.

[2]The garbage collector used in the .NET Runtime is discussed in Chapter 31, “Deeper into C#.”

[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).

Member Functions

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

Other member functions can be declared 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());
    }
}
```

In this example, the data fields are accessed directly. This is usually a bad idea, since 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, a property would be used, which gives the benefits of a member function while retaining the user model of a field. See Chapter 18, "Properties," for more information.

ref and out Parameters

Having to call two member functions to get the values may not always be convenient, so it would be nice to be able to get both values with a single function call. There's only one return value, however. The solution is to use reference (or `ref`) parameters, so that the values of the parameters passed into the member function can be modified:

```
// 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, but when compiled, it generates an error message that says that uninitialized values were used for the `ref` parameters `x` and `y`. This means that 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.

There are two ways around this. The first is to initialize the variables when they are 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 exactly like `ref` parameters except that an uninitialized variable can be passed to them, and the call is made with `out` rather than `ref`. [4]

[4] [4] From the perspective of other .NET languages, there is 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.

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.

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

```
    }
        // 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.

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.

Chapter 6: Base Classes And Inheritance

Overview

AS DISCUSSED IN CHAPTER 1, "Object-Oriented Basics," it sometimes 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 an `Engineer` and methods to handle billing for that `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`, except that 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 is 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

the call to the `base` class constructor was omitted, the compiler would call the base class constructor with no parameters.
`CivilEngineer` has a different way to calculate charges; the minimum charge is for 1 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.
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 is needed is a way to correctly identify the type of an engineer. This can be done 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 is code to check all the possible types and call the correct function. That's a lot of extra code, and it would be untenable if there were 50 kinds of engineers.

Worse is the fact 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, the example can be modified 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 that the base class has to know. It needs no knowledge of the derived types, other than to know that each derived class can implement `CalculateCharge()` and `TypeName()`, if desired. In the derived class, the functions are declared with the `override` keyword, which means that they are 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 is 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.

There is a small amount of overhead with the virtual dispatch, so it shouldn't be used 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.

[1] For a discussion of why this works this way, see Chapter 11, "Versioning Using New and Override."

Abstract Classes

There is a small problem with the approach used so far. 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, for example:

```
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 is needed is a way to force `ChemicalEngineer` to implement `TypeName()`.

This can be done 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 that 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` cannot 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.

```
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);
    }

    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);
    }
    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 changed by the addition of `abstract` before the class, which indicates that the class is abstract (i.e., 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`.

[2]A similar effect can be achieved by using interfaces. See Chapter 10, “Interfaces,” for a comparison of the two techniques.

Sealed Classes

Sealed classes are used to prevent a class from being used as a base class. It is primarily useful to prevent unintended derivation.

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

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

Chapter 7: Class Member Accessibility

Overview

ONE OF THE IMPORTANT DECISIONS to make when designing an object is how accessible to make the members. In C#, accessibility can be controlled 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 that 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, "Other Class Stuff."

`Internal` is a way of granting access to a wider set of classes without granting access to everybody, and it is 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 the use of friends, which provide access to a specific class. Friend provides greater granularity in specifying who can access a class, but in practice, the access provided by internal is usually sufficient.

In general, all classes should be `internal` unless users should be able to access them.

Using internal on Members

The `internal` modifier can also be used 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 10 drawing objects. It's valid for the user to have a reference to a `DrawingObject`, but it would be invalid for the user to call `Render()` for that object, so this is prevented by making the `Render()` function internal.

Tip This code doesn't make sense in a real program. The .NET Common Language Runtime has a number of collection classes that make this sort of thing much more straightforward and less error-prone.

internal protected

To provide some extra flexibility in how a class is defined, the `internal protected` modifier can be used 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.

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 does not exceed the accessibility of the class.

Consider the following situation:

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

If this class were declared as a public class, the accessibility of the members would be the same as the stated accessibility; i.e., `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`.

Chapter 8: Other Class Stuff

Overview

THIS CHAPTER DISCUSSES some miscellaneous issues dealing with classes, including constructors, nesting, and overloading rules.

Nested Classes

Sometimes, it is convenient to nest classes within other classes, such as when a helper class is only used by one other class. The accessibility of the nested class follows similar rules to the ones outlined 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 of the nested class. Just as a private field is always visible within a class, a private nested class is also visible 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 just 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 it 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 would be to make `Token` an `internal` class, so that it wouldn't be visible outside the assembly, but with this solution, it would still be visible inside the assembly.
The solution also misses out on an important benefit of using the nested class. A nested class makes it very clear to those reading the source 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.
Nesting can also be used as an organizational feature. If the `Parser` class were within a namespace named `Language`, you might require a separate namespace named `Parser` to nicely organize the classes for `Parser`, and that 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, 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 very important. In the .NET Runtime, the programmer can't control the destruction of objects, but it's helpful to know the areas that can be controlled.

Constructors

In C#, there is no default constructor created for objects. For classes, a default (e.g., parameterless) constructor may be written if needed.
A constructor can invoke a constructor of the base type by using the `base` syntax:

```
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.

```
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);
    }
}
```

Initialization

If the default value of the field isn't what is 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
{
     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.

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 that most people think of destructors, where the destructor is called when the object is deleted.

What is 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. This means that the programmer doesn't have direct control over the when the destructor is called, and it is therefore less useful than in languages such as C++. If cleanup is done in a destructor, there should also be another method that performs the same operation so the user can control the process directly.
For more information on this, see the section on garbage collection in Chapter 31, "Deeper into C#."

Overloading and Name Hiding

In C# classes—and in the Common Language Runtime in general—members are overloaded based upon the number and types of their parameters. They are not overloaded based upon the return type of the function. [1]

```
// error
using System;
class MyObject
{
    public string GetNextValue(int value)
    {
       return((value + 1).ToString());
    }
    public int GetNextVa ue(int value)
    {
       return(value + 1);
    }
}
class Test
{
    public static void Main()
    {
```

```
        MyObject my = new MyObject();
        Console.WriteLine("Next: {0}", my.GetNextValue(12));
    }
}
```

This code doesn't compile because the overloaded `GetNextValue()` functions differ only in return type, and the compiler can't figure out which function to call. It is therefore an error to declare functions that differ only by return type.

Name Hiding

In C#, method names are hidden based upon the name of the method, rather than upon the signature of the method. Consider the following example:

```
// error
using System;
public class Base
{
    public int Process(int value)
    {
        Console.WriteLine("Base.Process: {0}", value);
}
public class Derived: Base
{
    public int Process(string value)
    {
        Console.WriteLine("Derived.Process: {0}", value);
    }
}
class Test
{
    public static void Main()
    {
        Derived d = new Derived();
        d.Process("Hello");
        d.Process(12); // error
        ((Base) d).Process(12); // okay
    }
}
```

If the two overloaded `Process()` functions were in the same class, they would both be accessible. Because they are in different classes, the definition of `Process()` in the derived class hides *all* uses of that name in the base class.

To be able to access both functions, `Derived` would need to overload the version of `Process()` contained in the base class, and then forward the call to the base class's implementation.

[1] In other words, C++ covariant return types are not supported.

Static Fields

It is 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, the following could be used to track the number of instances of a class that were created.

```
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 instanced 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 either through 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 or through an instance.

Static Member Functions

The previous example exposes an internal field, which is usually something to be avoided. It can be restructured to use a static member function instead of a static field:

```
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 does the correct thing, and no longer exposes the field to users of the class, which increases future flexibility. Because it is a static member function, it is 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 18, “Properties.”

Static Constructors

Just as there can be other static members, there can be static constructors. A static constructor will be called before the first instance of an object is created and is useful to do setup work that needs to be done 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 only guarantees that it is called sometime after the start of the program and before the first instance of an object is created. This specifically means that it can’t be determined in the static constructor that an instance is about to be created.

A static constructor is declared simply by adding the `static` modifier in front of the constructor definition. A static constructor cannot have any parameters.

```
class MyClass
{
    static MyClass()
    {
    }
}
```

There is no static destructor analog of a destructor.

Constants

C# allows values 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 that its value can- not 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);
    }
}
```

readonly Fields

Because of the restriction on constant types being knowable at compile time, `const` cannot be used in many situations.
In a `Color` class, it can be very 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 static const Color Red = new Color(255, 0, 0);
    public static const Color Green = new Color(0, 255, 0);
    public static const Color Blue = new Color(0, 0, 255);
}
class Test
{
    static void Main()
    {
        Color background = Color.Red;
    }
}
```

This clearly doesn't work, since the static members `Red`, `Green`, and `Blue` can't be calculated at compile time. But making them normal public members doesn't work either, since anybody could change the red value to olive drab, or puce.
The `readonly` modifier is designed for exactly that situation. By applying `readonly`, the value can be set in the constructor or in an initializer, but can't be modified 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.

```
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 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 to use, 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.

Note The experienced C++ programmers are probably cringing at the last code example. It embodies one of the classical problems with the way C++ deals with memory management. Passing back an allocated object means that 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, because 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.

Private Constructors

Because there are no global variables or constants in C#, all declarations must be placed within a class. This sometimes leads to classes that are composed entirely of static members. In this case, there is no reason to ever instantiate an object of the class, and this can be prevented by adding a `private` constructor to the class.

```
// Error
using System;
class PiHolder
{
    private PiHolder() {}
    static double Pi = 3.1415926535;
}
class Test
{
    PiHolder pi = new PiHolder(); // error
}
```

Though the addition of `private` before the constructor definition does not change the actual accessibility of the constructor, stating it explicitly makes it clear that the class was intended to have a private constructor.

Variable-Length Parameter Lists

It is sometimes useful to define a parameter to take a variable number of parameters `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);
        for (int index = 0; index < args.GetLength(0); index++)
        {
```

```
            WriteString(args[index].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 is 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 line

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

is rewritten as

```
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 that 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 9: Structs (Value Types)

Overview

CLASSES WILL BE USED 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 is cheap and fast to allocate and doesn't have the overhead of references. In that case, a value type is used, which is done 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.

A Point Struct

In a graphics system, a value class could be used to encapsulate a point. Here's how it would be declared:

```
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;
}
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, there's a little bit of magic that goes on 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 (i.e., 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 is now a reference type, and we can treat it as if it were 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:

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 the cast could fail.
This code can be represented by Figure 9.1. 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.

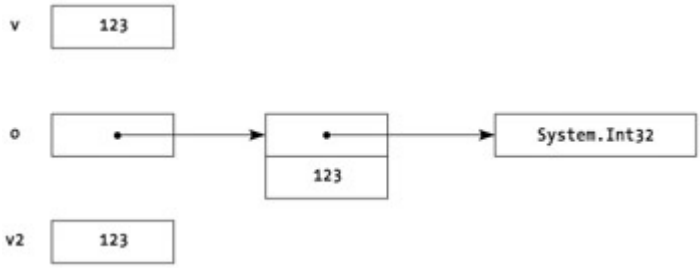


Figure 9-1. Boxing and unboxing a value type

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
```

Structs and Constructors

Structs and constructors behave a bit differently from 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 is 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.

It is 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

Structs should only be used for types that are really just a piece of data—for types that could be used in a similar way to the built-in types. A type, for example, 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 is not possible with value types.

Chapter 10: Interfaces

Overview

INTERFACES ARE CLOSELY RELATED to abstract classes; they resemble an abstract class that has 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 versions 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 ISclalable.ScaleX()
   public void ScaleX(float factor)
   {
      // scale the object here.
   }
      // implementing ISclalable.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 of the objects derive from `DiagramObject`, so that 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 that `TextObject` must have functions that match every function in the interface. Interface members have no access modifiers; the class that implements the interface sets the visibility of the interface member.

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 will discuss 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 ISclalable.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, the type is checked to make sure that 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 would be to restructure the code with exception handling, but that's not a great idea, because it would make the code more complex, and exception handling should generally be

reserved for exceptional conditions. It's also not clear whether it would 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, the loop can be rewritten 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 ISclalable.ScaleX()
   public void ScaleX(float factor)
   {
      Console.WriteLine("ScaleX: {0} {1}", text, factor);
      // scale the object here.
   }

      // implementing ISclalable.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 would 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 is not enough:

```
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. Here are 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 only be one base class in the .NET Runtime world, this decision is pretty important. If a base class is required, users will be very disappointed if they already have a base class and are unable to use the feature.

When using interfaces, remember that there is 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 will not 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() {}
```

```
}
```

That works fine if there are no name collisions between the functions in the interfaces. But if the example was 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()`?
It's ambiguous, so the compiler reports an error. If the user controlled either of the interfaces, the name in one of them could be changed, but that's not a great solution; why should `IFoo` have to change the name of its function just because `IBar` has the same name?
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:

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

This is what we 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 is 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 that the only way the interface can be accessed is by casting the object to the appropriate interface.
To expose one of the functions, a forwarding function is added 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()`.
This hiding can be used for other purposes, as detailed in the next section.

Implementation Hiding

There may be cases where it makes sense to hide the implementation of an interface from the users of a class, either because it's not generally useful, or just 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 only be accessed through the interface.

Interfaces Based on Interfaces

Interfaces can also be combined together to form new interfaces. The `ISortable` and `ISerializable` interfaces can be combined together, 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`.

Chapter 11: Versioning Using new and override

Overview

SOFTWARE PROJECTS RARELY EXIST as a single version of code that is 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 harder things to do in software. One reason why it's tough is that it requires a bit of planning and foresight; the areas that might change have to be determined, and the design must be modified 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, some revisions can be made 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, though installing a new version of a library may cause other applications that use an older version of the library to cease functioning.

Managed environments that don't expose class member or layout information in the metadata fare better at versioning but it's still possible to write code that versions poorly.

A Versioning Example

The following code presents a simple versioning scenario. The program uses a class named `Control`, which is provided by another company.

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

During implementation of `MyControl`, the virtual function `Foo()` is added:

```
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() {}
}
```

That `Control` uses `Foo()` as the name of the function 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`.
Which 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 parameters and "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 run- time behavior is unchanged.
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 specify that the default behavior of not considering the function an override continue, the `new` modifier is added 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, the `override` modifier is used.

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

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

This tells the compiler that the function really is an override.

Caution About this time, there's somebody in the back of the room who's thinking, "I'll just put new on all of my virtual functions, and then I'll never have to deal with the situation again." Doing so is discouraged because it reduces the value that the new annotation has to somebody reading the code. If new is only used when it is 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 the new has meaning.

Chapter 12: Statements and Flow of Execution

Overview

THE FOLLOWING SECTIONS DETAIL the different statements that are available within the C# language.

Selection Statements

The selection statements are used to 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 is just too easy to inadvertently omit a `break` statement at the end of a `case`, or not to notice that there is 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`.

```
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");
      }
   }
}
```

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 is 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]

[1]The actual number is determined based upon the performance tradeoffs of each method.

[2]If you're unfamiliar with hashing, consider looking at the `System.Collections.HashTable` class or a good algorithms book.

Iteration Statements

Iteration statements are often known as looping statements, and they are used to 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++;
      }
   }
}
```

The `break` statement may be used to exit the `while` loop, and the `continue` statement may be used 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 will generate 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 may be used to control the flow of execution in the loop.

For

A `for` loop is used to iterate over several values. The loop variable may be declared 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 cannot be accessed outside of 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 may be used to control the flow of execution in the loop.

Foreach

This is a very 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 because 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 is 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, such as Perl, provide a different construct for dealing with this problem, and C# also provides such a construct. The preceding example can be rewritten 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);
        }
    }
}
```

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 only store values of type `object`. `Foreach` also works for objects other than arrays. In fact, it works for any object that implements the proper interfaces. It can, for example, be used to 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 the "Indexers and Foreach" section in Chapter 19, "Indexers," for more information.

The one thing that can't be done in a `foreach` loop is changing the contents of the container. 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.

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 is used to break out of the current iteration or `switch` statement, and continue execution after that statement.

Continue

The `continue` statement skips all of the later lines in the current iteration statement, and then continues executing the iteration statement.

Goto

The `goto` statement can be used to jump directly to a label. Because the use of `goto` statements is widely considered to be harmful,[3] C# prohibits some of their worst abuses. A goto cannot 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 of 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.

[3]See "GO TO considered harmful," by Edsger W. Dijkstra, at `http://www.net.org/html/history/detail/1968-goto.html`

Definite Assignment

Definite assignment rules prevent the value of an unassigned variable from being observed. Suppose the following is written:

```
// 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, operations cannot be done with a class variable before it is 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 definite assignment is considered. The runtime will always make sure they're zeroed out, but the compiler will still check to be sure that they're initialized to a value before they're used.

A struct is initialized either through a call to a constructor or by setting all the members of an instance before it is 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}, {0})", 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, `myNumber1` is initialized by the call to `new`. Remember that for structs, there is no default constructor, 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. This can obviously only be done 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 be initialized.
For example, suppose there is 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 13: Local Variable Scoping

Overview

IN C#, LOCAL VARIABLES CAN ONLY be given names that allow them to be uniquely identified in a given scope. If a name has more than one meaning in a scope and there is no way to disambiguate the name, the innermost declaration of the name is an error and must be changed. 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 that it must be an instance variable.

The preceding construct is preferred to renaming the constructor parameters or member variables to avoid the name conflict.

In the following situation, there is no way to name both variables, and the inner declaration is therefore an error:

```
// error
using System;
class MyObject
{
    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 *were* multiple versions—inside a nested scope.

Chapter 14: Operators

Overview

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. The default precedence can be changed by grouping elements with parentheses.

```
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 that operations are performed left to right, except for the assignment and conditional (?:) operators, which are performed right to left.

The following table summarizes all operators in precedence from highest to lowest.

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	| |
Conditional	?:
Assignment	= *= /= %= += -= <<= >>= &= ^= |=

Built-In Operators

For numeric operations in C#, there are built-in operators for the `int`, `uint`, `long`, `ulong`, `float`, `double`, and `decimal` types. Because there aren't built-in operators for other types, expressions must first be converted to one of the types for which there is an operator before the operation is performed. This means that when operations are done with numeric types that can be converted implicitly to `int`–those types that are "smaller" than `int`–the result will have to be cast to store it in the same type.

```
// 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
   }
}
```

User-Defined Operators

User-defined operators may be declared for classes, and they function in the same manner in which the built-in operators function. See Chapter 25, "Operator Overloading," for more information.

Numeric Promotions

See Chapter 15, "Conversions," 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 very specific rules that they follow;[1] for full details, see the CLR. If executed in a checked context, arithmetic expressions on non-floating types may throw exceptions.

Unary Plus (+)

For unary plus, the result is simply the value of the operand.

Unary Minus (-)

Unary minus only works on types for which there is a valid negative representation, and it returns the value of the operand subtracted from zero.

Addition (+)

In C#, the `+` sign 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. This is demonstrated by the following code:

```
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

String concatenation can be performed between two strings, or between a string and an operand of type `object`. [2] If either operand is null, an empty string is substituted for that operand.
Operands that are not of type `string` will be automatically be converted to a string by calling the virtual `ToString()` method on the object.

Subtraction (-)

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 (*)

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 (/)

The first operand is divided by the second operand. If the second operand is zero, a `DivideByZero` exception is thrown.

Remainder (%)

The result `x % y` is computed as `x-(x / y) * y` using integer operations. If y is zero, a `DivideByZero` exception is thrown.

Shift (<< and >>)

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 zero if `x` is non-negative, and 1 if `x` is negative.

Increment and Decrement (++ and --)

The increment operator increases the value of a variable by 1, and the decrement operator decreases the value of the variable by 1. [3]

Increment and decrement can either be used as a prefix operator, where the variable is modified before it is read, or as a postfix operator, where the value is returned before it is modified.

For example:

```
int  k = 5;
int  value = k++;    // value is 5
     value = --k;    // value is still 5
     value = ++k;    // value is 6
```

[1]They conform to IEEE 754 arithmetic.

[2]Since any type can convert to object, this means any type.

[3]In `unsafe` code, pointers increment and decrement by the size of the pointed-to object.

Relational and Logical Operators

Relational operators are used to compare two values, and logical operators are used to perform bitwise operations on values.

Logical Negation (!)

The `!` operator is used to return the negation of a Boolean value.

Relational Operators

C# defines the following relational operations:

OPERATION	ESCRIPTION
`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. 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 are the same object, not whether they have the same value.

For value types, the process is the same, except that the built-in relational operator for value types compares each of the fields in the struct, and returns true if all the values are identical.
For the `string` type, the relational operators are overloaded so that `==` and `!=` compare the values of the strings, not the references.

Logical Operators

C# defines the following logical operators:

OPERATOR	ESCRIPTION
`&`	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 expression

a && b
`b` is only evaluated if `a` is true. In the expression

a || b
`b` is only evaluated if `a` is false.

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 is true, the value of the operator is the first expression following the question mark, or `15` in this case. If the control expression is false, the value of the operator is the expression following the colon, or `5`.

Assignment Operators

Assignment operators are used to assign a value to a variable. There are two forms: the simple assignment and the compound assignment.

Simple Assignment

Simple assignment is done in C# using the single equals "=". 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 compound operators are the following:

+= -= *= /= %= &= = ^= <<= >>=

The compound operator

x <op>= y

is evaluated exactly as if it were written as

x = x <op> y

with two exceptions:

- x is only evaluated once, and that evaluation is used for both the operation and the assignment.
- If `x` contains a function call or array reference, it is only performed 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 is not implicitly converted to a `short`. In this case however, because `short` can be implicitly converted to `int`, and it is possible to write

x = 3;

the operation is permitted.

Type Operators

Rather than dealing with the values of an object, the type operators are used to 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 the `type` object is obtained for a type, it can be queried using reflection to obtain information about the type. See the section titled "Deeper Reflection" in Chapter 31, "Deeper into C#," for more information.

is

The `is` operator is used to determine 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 very 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 or interface. If the object can't be converted to that type or interface, the operator returns null. Using `as` is more efficient than the `is` operator, since the `as` operator only needs to check the type of the object 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, these lines

```
        if (o is IAnnoy)
        {
```

```
        IAnnoy annoyer = (IAnnoy) o;
        annoyer.PokeSister(sister);
    }
```

could be replaced with these:

```
    IAnnoy annoyer = o as IAnnoy;
    if (Annoyer != null)
        annoyer.PokeSister(sister);
```

Chapter 15: Conversions

Overview

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 that 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.

[1]Conversions from `int`, `uint`, or `long` to `float` and from `long` to `double` may result in a loss of precision, but will not result in a loss of magnitude.

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 is 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`.

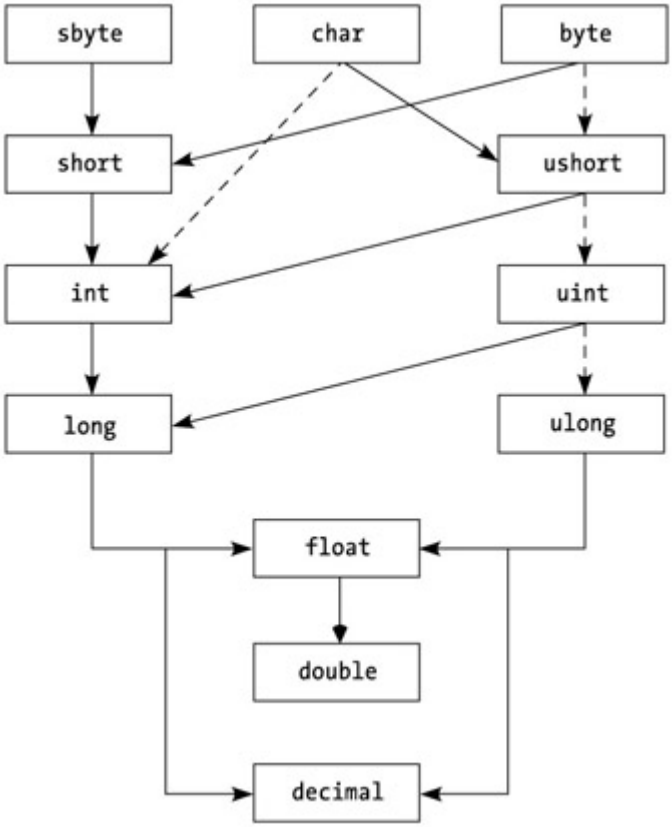


Figure 15-1. C# conversion hierarchy

Note that the path taken from a source type to a destination type in the figure does not represent how the conversion is done; it merely indicates that it can be done. In other words, the conversion from `byte` to `long` is done 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;
   }
}
```

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 only match the `int` parameter to 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 wasn't there, it would 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 is a single-arrow implicit conversion to a signed type, it will be preferred over all conversions to unsigned types. This is graphically represented in Figure 15-1 by the dotted arrows; the compiler will choose a single solid arrow over any number of dotted arrows.
This rule only applies for the case where there is a single-arrow conversion to the signed type. If the function that took a `short` was changed to take an `int`, there would be no "better" conversion, and an ambiguity error would be reported.

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, and therefore converting from `long` to `short` is an explicit conversion.

Viewed another way, an explicit numeric conversion may result in a value that is different than 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 is depending on this behavior.

Checked Conversions

In other cases, it may be useful to check whether the conversion succeeded. This is done 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 is done in a `checked` context, if the source value will not 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 or not is determined at compile time, and the checked state does not 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 be used to indicate that none of the conversions in a block should ever be checked for conversions.

It is sometimes useful to be able to specify the checked state for a single statement; in this case, the `checked` or `unchecked` operator can be specified 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.

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 does *not* 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`; this is shown because when the virtual function `WhoAmI()` is called, the version from `Derived` is called. It is 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, "Object-Oriented Basics," 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 there is a "could-be" relationship. 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 (in the previous example) will be checked to see if it is 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 shows the previous example pictorially.

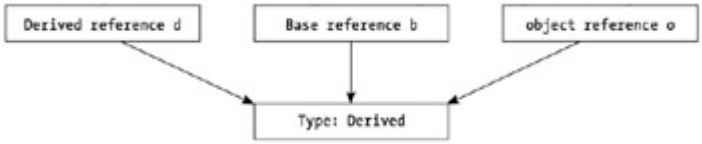


Figure 15-2. Different references to the same instance

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 is known at compile time that it works.

Once again, the conversion to an interface does not 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 back from the interface to an object is an operation that is 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 the common case. 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:

```
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: {0}\nRequests: {1}\nFailures; {0}\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);
  }
}
```

In this example, there are 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, and 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 inter- face that it is based upon. It can be converted explicitly to a reference to any interface that it isn't based upon. This would be successful only if the interface reference was a reference to an object that implemented 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 is no explicit conversion from an interface to a struct.

Chapter 16: Arrays

Overview

ARRAYS IN C# ARE reference objects; they are 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 does not contain boxed instances.

Arrays are declared using the following syntax:

<type>[] identifier;;
The initial value of an array is null. An array object is created 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` with the value `null`.

Array Initialization

Arrays can be initialized at the same time as they are created. During initialization, the `new int[x]` can be omitted, 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, either a multidimensional or jagged array can be used.

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 5, 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 4, 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 <= 2). When the matrix array is allocated, a single chunk is obtained from the heap to store the entire array. It can be represented by Figure 16-1.

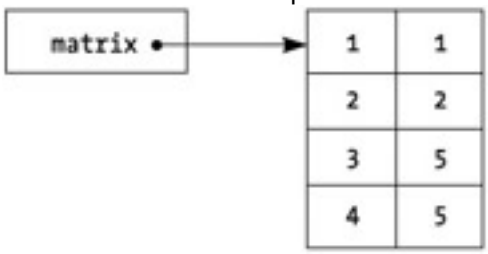


Figure 16-1. Storage in a multidimensional array

Jagged Arrays

A jagged array is merely an array of arrays and is called a “jagged” array because it doesn’t have to be square. For example:

int[][] matrix = new int[3][];

matrix[0] = new int[10];

matrix[1] = new int[11];

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 3 elements. Its elements are integer arrays. The first element is an array of 10 integers, the second is an array of 11 integers, and the third is an array of 2 integers.

Since the elements of jagged arrays are arrays, when the top-level jagged array is allocated, each element is initialized to null. Therefore, each element must be initialized to a valid array. Because of this, there is no initialization syntax for the elements of a jagged array. In the two-dimensional case, however, the preceding code can be rewritten as:

int[][] matrix = {new int[5], new int[4], new int[2]};

matrix[0][3] = 4;

matrix[1][1] = 8;

matrix[2][0] = 5;

This array could be represented by Figure 16-2. The `matrix` variable is a reference to an array of 3 references to arrays of integers. Four heap allocations were required for this array.

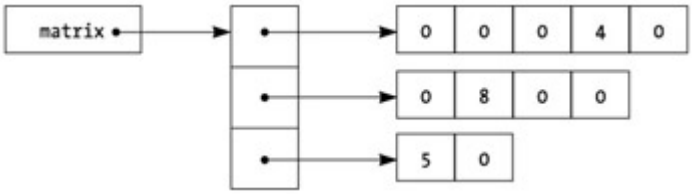


Figure 16-2. Storage in a jagged array

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.

The class can be rewritten 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 us to create an instance and 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 would need to do the same initialization for each element, and if that wasn't the right initialization, all of 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, the elements of `S` have an implicit reference conversion to the element type of `T`, and both `S` and `T` are reference types. In other words, if there is an array of class references, 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`.

System.Array Type

Because arrays in C# are based on the .NET Runtime `System.Array` type, there are several operations that can be done with them that aren't traditionally supported by array types.

Sorting and Searching

The ability to do sorting and searching 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.
These functions work for the built-in types. To enable them for user-defined classes or structs, see Chapter 27, "Making Friends with the .NET Frameworks."

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: Strings

Overview

ALL STRINGS IN C# are instances of the `System.String` type in the Common Language Runtime. Because of this, there are many built-in operations 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 that the characters contained in the string cannot be modified by users of the string. All operations that are performed by the `string` class return a modified version of the string rather than modifying the instance on which the method is called.

The `String` class supports the following comparison and searching 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
`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 following 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()`	Left aligns a string in field
`PadRight()`	Right 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 lower-case version of a string
`ToUpper()`	Returns an upper-case version of a string
`Trim()`	Removes white space 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

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; a different representation may be obtained by calling `String.Format()`. See the section on formatting in Chapter 30, ".NET Frameworks Overview," for more information.

An Example

The split function can be used 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. This can be fixed by using the regular expression classes.

StringBuilder

Though the `String.Format()` function can be used to create a string based on the values of other strings, it isn't the most efficient way to assemble strings. The run-time provides the `StringBuilder` class to make this process easier.

The `StringBuilder` class supports the following properties and methods:

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`

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 the `StringBuilder` class can be used 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 very powerful method for doing search and/or replace functions.

While there are a few examples of using regular expressions in this section, a detailed description of them is beyond the scope of the book. There are several regular expression books available, and the subject is also covered in most books about Perl.

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]

The previous example using `Split()` can be revised 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.

```
// file: regex.cs
// compile with: csc /r:system.text.regularexpressions.dll 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 = new char[] {' ', ','};
      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.

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: logparss e.cs
// compile with: csc logparse.cs /r:system.net.dll /
r:system.text.regularexpressions.dll
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.FileExists(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.Group("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 non-white
                @"(?<method>\S+)\s" +
                    // match any non-white
                @"(?<uri>\S+)\s" +
                    // match any non-white
                @"(?<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.Group("time"));
                Console.WriteLine("IP Address: {0}",
                            match.Group("ip"));
                Console.WriteLine("Method: {0}",
                            match.Group("method"));
                Console.WriteLine("Status: {0}",
                            match.Group("status"));
                Console.WriteLine("URI: {0}\n",
                            match.Group("uri"));
            }
        }
        f.Close();
      }
   }
}
```

The general structure of this code should be familiar. There are two regular expressions used in this example. The date string and the regular expression used to match it are the following:

```
#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 are broken down into separate elements. This

```
\:
```

matches the colon (`:`). The backslash (`\`) is required because the colon by itself means something else. This

```
\s
```

matches a single character of whitespace (tab or space). In this next part

```
(?<date>[^\s]+)
```

the `?<date>` names the value that will be matched, so it can be extracted later. The `[^\s]` is called a character group, with the `^` character meaning "none of the following characters." This group therefore matches any non-whitespace character. Finally, the `+` character means to match one or more occurrences of the previous description (non-whitespace). The parentheses are used to delimit how to match the extracted string. In the preceding example, this part of the expression matches `1999-12-31`. To match more carefully, the `/d` (digit) specifier could have been used, with the whole expression written as:

```
\:\s(?<date>\d\d\d\d-\d\d-\d\d)\s
```

That covers the simple regular expression. A more complex regular expression is used to match each line of the log file. Because of the regularity of the line, `Split()` could also have been used, 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 non-whitespace for method
(?<uri>\S+)\s          // any non-whitespace for uri
(?<status>\d+)         // any digit for status
```

[1]The program is written using the .NET intermediate language—the same one that C# produces as output from a compilation.

Chapter 18: Properties

Overview

A FEW MONTHS AGO, I was writing some code, and I came up with a situation where one of the fields in a class (`Filename`) could be derived from another (`Name`). I 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. I 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.
I also had to remember that when I wanted to get the filename, I had to call the `getFilename()` member function to get it, rather than merely referring to the filename member of the class. This makes the model a bit tougher to grasp; instead of `Filename` just being a field, I have to remember that I'm really calling a function whenever I need to access it.

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 member function 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 Base Class Library; 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—that handle getting or setting the property. Here's a simple example:

```
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 that the property should be set to. 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 only has a getter is called a read-only property, and a property that only has 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 is not 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 in cases where the property is derived from other fields, and also to do lazy initialization and only fetch a value if the user really needs it.

Suppose that a car maker 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 –, and the expensive operation of calculating them is deferred until it is actually needed.

Side Effects When Setting Values

Properties are also very 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.
This interaction could also be implemented more generally using events, which are covered in Chapter 23, “Events.”

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 cannot be declared with the `virtual`, `abstract`, or `override` modifiers.
When `readonly` fields were discussed Chapter 8, "Other Class Stuff," there was a case that initialized some static readonly fields. The same thing can be done 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 is cheap to create or it is 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, let's 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 there would normally be a field access. However, there is no reason that the underlying runtime environment can't inline the accessors as it would any other simple function, so there is often[1] no performance penalty in choosing a property instead of a simple field. The opportunity to be able to revise the implementation at a later time without changing the interface can be invaluable, so properties are usually a better choice than fields for public members.

There does remain a small downside of 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.

[1]The Win32 version of the .NET Runtime does perform the inlining of trivial accessors, though other environments wouldn't have to.

Chapter 19: Indexers

Overview

IT SOMETIMES MAKES SENSE to be able to index an object as if it were an array. This can be done by writing an indexer for the object, which can be thought of as a smart array. As 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));
    }

        public object this[int column]
        {
            get
            {
                return(row[column - 1]);
            }
            set
            {
                row[column - 1] = value;
            }
    }
    ArrayList row;
}

class Test
{
    public static void Main()
    {
        DataRow row = new DataRow();
        row.Load();
        DataValue val = (DataValue) row[0];
        Console.WriteLine("Column 0: {0}", val.Data);
        val.Data = 12; // set the ID
    }
}
```

The `DataRow` class has functions to load in a row of data, functions to save the data, and an indexer function to provide access to the data. In a real class, the `Load()` function would load data from a database.

The indexer function is written the same way that a property is written, except that it takes an indexing parameter. The indexer is declared using the name `this` since it has no name.

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 object this[int column]
    {
        get
        {
            return(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);
        }
        return(-1);
    }
    public object this[string name]
    {
        get
        {
            return this[FindColumn(name)];
        }
        set
        {
            this[FindColumn(name)] = value;
        }
    }
    ArrayList row;
}
class Test
{
    public static void Main()
    {
        DataRow row = new DataRow();
        row.Load();
```

```
        DataValue val = (DataValue) row["Id"];
        Console.WriteLine("Id: {0}", val.Data);
        Console.WriteLine("Salary: {0}",
        ((DataValue) row["Salary"]).Data);
        ((DataValue)row["Name"]).Data = "Barney"; // set the name
        Console.WriteLine("Name: {0}", ((DataValue) 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.

Indexers can have more than one parameter, to simulate a multidimensional virtual array.

Indexers and `foreach`

If an object can be treated as an array, it is often convenient to iterate through it using the **`foreach`** statement. To enable the use of **`foreach`**, and similar constructs in other .NET languages, the **`IEnumerable`** interface must be implemented by the object. This interface has a single member named **`GetEnumerator()`**, which returns a reference to an **`IEnumerator`** interface, which has member functions that are used to perform the enumeration.

The **`IEnumerator`** interface can be implemented directly by the container class, or it can be implemented by a private class. Private implementation is preferable since it simplifies the collection class.

The following example expands the previous example to enable **`foreach`** :

```
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: IEnumerable
    {
        class DataRowEnumerator: IEnumerator
        {
            public DataRowEnumerator(DataRow dataRow)
            {
                this.dataRow = dataRow;
                index = -1;
            }
            public bool MoveNext()
            {
                index++;
                if (index >= dataRow.row.Count)
                    return(false);
                else
                    return(true);
            }
            public vo d Reset()
            {
                index = -1;
            }
            public object Current
            {
                get
                {
                    return(dataRow.row[index]);
                }
            }
            DataRow dataRow;
            int index;
    }
    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 object this[int column]
{
    get
    {
        return(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);
    }
    return(-1);
}
public object this[string name]
{
    get
    {
        return this[FindColumn(name)];
    }
    set
    {
        this[FindColumn(name)] = value;
    }
}
    public IEnumerator GetEnumerator()
    {
        return((IEnumerator) new DataRowEnumerator(this));
```

```
}
ArrayList row;
}
class Test
{
   public static void Main()
   {
      DataRow row = new DataRow();
      row.Load();
      foreach (DataValue dataValue in row)
      {
         Console.WriteLine("{0}: {1}",
            dataValue.Name, dataValue.Data);
      }
   }
}
```

The `foreach` loop in `Main()` is rewritten by the compiler as follows:

```
IEnumerator enumerator = row.GetEnumerator();
while (enumerator.GetNext())
{
   DataValue dataValue =
      (DataValue) enumerator.Current;
   Console.WriteLine("{0}: {1}",
      dataValue.Name, dataValue.Data)
}
```

Design Guidelines

Indexers should be used only in situations where the abstraction makes sense. This usually depends on whether the object is a container for some other object.

Indexers should have both a getter and a setter, as arrays are read/write objects. If the indexer only has a getter, consider replacing it with a method.

Chapter 20: Enumerators

Overview

ENUMERATORS ARE USEFUL WHEN a value in the program can only have a specific set of values. A control that could only be one of four colors, or a network package that supports only two protocols, are situations where an enumeration can improve 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,
    }


    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.

While enums do prevent the accidental specification of values outside of the enum range, the values that can be specified for an enum are not 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 that it is in the range of valid values. The `Draw` class throws an invalid argument exception if the argument is invalid.

Enumerator Base Types

Each enumerator has an underlying type that specifies how much storage is allocated for that enumerator. The valid base types for enumerators are `byte`, `sbyte`, `short`, `ushort`, `int`, `uint`, `long`, and `ulong`. If the base type is not 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 only depend 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:

```
[Flags]
enum BitValues
{
    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 would make sense to allow the user to `OR` several bits together, which wouldn't make sense for non-flag 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 the literal `0` can be converted to an enum type without a cast. This is allowed so that the following code can be written:

```
public void DoSomething(BitValues bv)
{
    if (bv == 0)
    {


    }
}
```

The `if` statement would have to be written as

```
if (bv == (BitValues) 0)
```

if this exception wasn't present. 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:

```
if (bv == (CornSoft.PlotLibrary.Drawing.LineStyle.BitValues) 0)
```

That's a lot of typing.

Chapter 21: Attributes

Overview

IN MOST PROGRAMMING LANGUAGES, some information is expressed through declaration, and other information is expressed through code. For example, in the following class member declaration

```
public int Test;
```

the compiler and runtime will reserve space for an integer variable and set its protection so that it is visible everywhere. This is an example of declarative information; it's nice because of the economy of expression and because the compiler handles the details for us.

Typically, the types of declarative information are predefined by the language designer and 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 like C++, a macro might be defined that stores the information in a field that is 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 that are placed on elements of source code, such as classes, members, parameters, etc. Attributes can be used 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. A use for the conditional attribute would look like this:

```
using System.Diagnostics
class Test
{
   [Conditional("DEBUG")]
   public void Validate()
   {
   }
}
```

Most programmers will use predefined attributes much more often than writing an attribute class.

Using Attributes

Suppose that for a project that a group was doing, it was important to keep track of the code reviews that had been performed on the classes so that it could be determined when code reviews were finished. The code review information could be stored in a database, which would allow easy queries about status, or it could be stored in comments, which would make it easy to look at the code and the information at the same time.
Or an attribute could be used, which would enable both kinds of access. To do that, an attribute class is needed. An attribute class defines the name of an attribute, how it can be created, and the information that will be stored. The gritty details of defining attribute classes will be covered in the section entitled "An Attribute of Your Own."

The attribute class will look 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 only be placed 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 `Comment`.
When the compiler comes to the attribute usage on 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 on a class.

Then, it checks to see if there is a constructor that matches the parameters we'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 there are named parameters, 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, that's what happens *logically*. In actuality, it only *looks* like it happens that way; see the "Attribute Pickling" sidebar for a description of how it is implemented.

A Few More Details

Some attributes can only be used once on a given element. Others, known as multi- use attributes, can be used more than once. This 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 multi-use.

In most cases, it's clear that the attribute applies to a specific program element. However, consider the following case:

```
class Test
{
  [MarshalAs(UnmanagedType.LPWSTR)]
  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?

There are several situations in which this can happen:

- 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, there is a case that is much more common than the other case, and it becomes the default case. To specify an attribute for the non- default case, the element the attribute applies to must be specified:

```
class Test
{
    [return:ReturnsHResult]
    public void Execute() {}
}
```

The `return`: indicates that this attribute should be applied to the return value.

The element may be specified even if there is no ambiguity. The identifiers are as follows:

SPECIFIER	DESCRIPTION
`assembly`	Attribute is on the assembly
`module`	Attribute is on the module
`type`	Attribute is on a class or struct
`method`	Attribute is on a method
`property`	Attribute is on a property
`event`	Attribute is on an event
`field`	Attribute is on a field
`param`	Attribute is on a parameter
`returnValue`	Attribute is on the return value

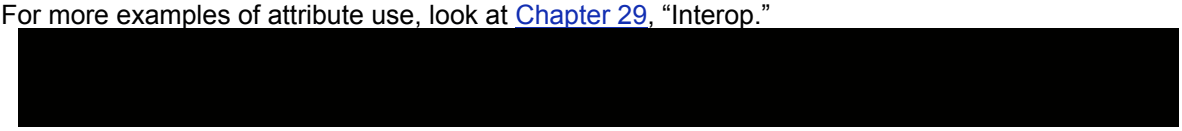
Attributes that are applied to assemblies or modules must occur after any `using` clauses and before any code.

```
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 is 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, look at Chapter 29, "Interop."



Attribute Pickling

There are a few reasons why it doesn't really work the way it's described, and they're 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 we'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.

An Attribute of Your Own

To define attribute classes and reflect on them at runtime, there are a few more issues to consider. This section will discuss some things to consider when designing an attribute.

There are 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 and are as follows:

177

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
`Return`	The method return value
`Delegate`	A delegate
`All`	Anywhere
`ClassMembers`	Class, Struct, Enum, Constructor, Method, Property, Field, Event, Delegate, Interface

As part of the `AttributeUsage` attribute, one of these can be specified or a list of them can be ORed together.
The `AttributeUsage` attribute is also used to specify whether an attribute is single- use or multi-use. This is done with the named parameter `AllowMultiple`. Such an attribute would look 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 is required for every use, and the optional items.

The information that is 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 only supports a subset of all the .NET Runtime types, and therefore, only some types can be used 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 non-public)
- A one-dimensional array of one of the above types

Reflecting on Attributes

Once attributes are defined on some code, it's useful to be able to find the attribute values. This is done 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)]
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
{
}

class Test
{
    public static void Main()
    {
        System.Reflection.MemberInfo info
        info = typeof(Complex);
        object[] atts;
        atts = info.GetCustomAttributes(typeof(CodeReviewAttribute))
        if (atts.GetLength(0) != 0)
        {
            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 loads all the attributes that are of the `CodeReviewAttribute` type. If the array of attributes has any entries, the first element is cast to a `CodeReviewAttribute`, and then the value is printed out. There can only be one entry in the array because `CodeReviewAttribute` is single-use.

This example produces the following output:

```
Reviewer: Eric
Date: 01-12-2000
Comment: Bitchin' Code
```

`GetCustomAttribute()` can also be called without a type, to get all the custom attributes on an object.

Note The “CustomAttributes" in the preceding example refers to attributes that are stored in a general attribute part of the metadata for an object. Some .NET Runtime attributes are not stored as custom attributes on the object, but are converted to metadata bits on the object. Runtime reflection does not support viewing these attributes through reflection. This restriction may be addressed in future versions of the runtime.

Chapter 22: Delegates

Overview

DELEGATES ARE SIMILAR TO interfaces, in that they specify a contract between a caller and an implementer. Rather than specifying an entire interface, a delegate merely specifies the form of a single function. Also, interfaces are created at compile time, whereas delegates are created at runtime.

Using Delegates

The specification of the delegate determines the form of the function, and to create an instance of the delegate, one must use a function that matches that form. Dele- gates are sometimes referred to as “safe function pointers.” Unlike function pointers, however, delegates in C# can call more than one function; if two delegates are added together, a delegate that calls both delegates is the result.

Because of their more 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. The sorting could be controlled 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 (all delegates must be static functions) as part of the `Employee` class, with each function describing a different kind of ordering.

When the container needs to be sorted, a delegate can be passed in 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 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 would be nicer if they didn't have to do that, and that can be done 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 isn't nice, however, is that the delegate is *always* created, even if it is never used. This is a bit wasteful. It would be better if the delegate were created on the fly, as needed. This can be done by replacing the static functions 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 private static delegate members `SortByName` and `SortById`, and the property created the static member if it hadn't been needed before. This would work well if the creation of the delegate were somewhat costly, and the delegate was 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 only for illustration. To implement a collection class that does sorting, the techniques used by the Frameworks are much easier. See Chapter 27, "Making Friends with the

Chapter 23: Events

Overview

A CLASS CAN USE AN EVENT to notify another class (or other classes) that something has happened. Events use the “publish-subscribe” idiom; a class publishes the events that it can raise, and classes that are interested in a specific event can subscribe to the event.

Events are often used in graphical user interfaces for notification that the user has made a selection, but they are well suited for any asynchronous operation, such as a file being changed, or an email message arriving.
The routine that an event will call is defined by a delegate. To make events easier to deal with, the design convention for events says that the delegate always takes two parameters. The first parameter is the object that raised the event, and the second parameter is an object that contains the information about the event. This object is always derived from the `EventArgs` class.

A New Email Event

Here’s an example of events.

```
using System;
class NewEmailEventArgs: EventArgs
{
    public NewEmailEventArgs(string subject, string message)
    {
        this.subject = subject;
        this.message = message;
    }
    public string Subject
    {
        get
        {
            return(subject);
        }
  }
    public string Message
    {
        get
       {
          return(message);
       }
    }
    string subject;
    string message;
}
class EmailNotify
{
    public delegate void NewMailEventHandler(object sender,
      NewEmailEventArgs e);
    public event NewMailEventHandler OnNewMailHandler;
```

```
    protected void OnNewMail(NewEmailEventArgs e)
    {
   if (OnNewMailHandler != null)
      OnNewMailHandler(this, e);
    }
    public void NotifyMail(string subject, string message)
    {
   NewEmailEventArgs e = new NewEmailEventArgs(subject, message);
   OnNewMail(e);
    }
}
class MailWatch
{
    public MailWatch(EmailNotify emailNotify)
    {
   this.emailNotify = emailNotify;
   emailNotify.OnNewMailHandler +=
         new EmailNotify.NewMailEventHandler(IHaveMail);
    }
    void IHaveMail(object sender, NewEmailEventArgs e)
    {
      Console.WriteLine("New Mail: {0}\n{1]",
            e.Subject, e.Message);
    }
    EmailNotify emailNotify;
}
class Test
{
   public static void Main()
   {
      EmailNotify emailNotify = new EmailNotify();
      MailWatch mailWatch = new MailWatch(emailNotify);
      emailNotify.NotifyMail("Hello!", "Welcome to Events!!!");
   }
}
```

The `NewEmailEventArgs` class contains the information that is passed when the `NewEmail` event is raised.

The `EmailNotify` class is responsible for handling the event; it declares the delegate that defines what parameters are passed when the event is raised, and it also defines the event itself. The `OnNewMail()` function is used to raise the event, and the helper function `NotifyMail()` takes the event information, packages it up into an instance of `NewEmailEventArgs`, and calls `OnNewMail()` to raise the event.

The `MailWatch` class is a consumer of the `EmailNotify` class. It takes an instance of the `EmailNotify` class and hooks the `IHaveMail()` function to the `OnNewMailHandler` event. Finally, the `Main()` function creates instances of `EmailNotify` and `MailWatch`, and then calls the `NotifyMail()` function to raise the event.

The Event Field

In the previous example, the event field is `EmailNotify.OnNewMailHandler`. Within the class that contains the event field, there are no restrictions on the usage.
Outside the declaration of `EmailNotify`, however, an event field can only be used on the left-hand side of the `+=` and `-=` operations; the field cannot be otherwise examined or modified.

Multicast Events

Events in C# are multicast, which means that raising an event may call multiple delegates with the event information. The order in which the delegates are called is not defined, and if one delegate throws an exception, it may result in other delegates not being called.

Sparse Events

Most classes will implement events using event fields, as done in the earlier example. If a class implements numerous events, but only a small fraction of them are likely to be used at once, reserving a separate field for each event can be wasteful of space. This might happen with a user interface control that supports lots of events.

In this situation, a class can declare event properties instead of event fields, and use a private mechanism for storing the underlying delegates. The following example revises the previous example, using event properties instead of event fields.

```
using System;
using System.Collections;
class NewEmailEventArgs: EventArgs
{
    public NewEmailEventArgs(string subject, string message)
    {
        this.subject = subject;
        this.message = message;
    }
    public string Subject
    {
        get
        {
            return(subject);
        }
    }
    public string Message
    {
        get
        {
            return(message);
        }
    }
    string subject;
    string message;
}
class EmailNotify
```

```
{
    public delegate void NewMailEventHandler(object sender,
       NewEmailEventArgs e);

    protected Delegate GetEventHandler(object key)
    {
       return((Delegate) handlers[key]);
    }
    protected void SetEventHandler(object key, Delegate del)
    {
       handlers.Add(key, del);
    }
    public event NewMailEventHandler   OnNewMailHandler
    {
       get
       {
           return((NewMailEventHandler)
                 GetEventHandler(onNewMailKey));
       }
       set
       {
            SetEventHandler(onNewMailKey, value);
       }
    }

    public void OnNewMail(NewEmailEventArgs e)
    {
          if (OnNewMailHandler != null)
             OnNewMailHandler(this, e);
    }
    public void NotifyMail(string subject, string message)
    {
          NewEmailEventArgs e = new NewEmailEventArgs(subject, message);
          OnNewMail(e);
    }
    Hashtable handlers = new Hashtable();
          // unique key for this event
    static readonly object onNewMailKey = new object();
}
class MailWatch
{
    public MailWatch(EmailNotify emailNotify)
    {
        this.emailNotify = emailNotify;
```

```
        emailNotify.OnNewMailHandler +=
        new EmailNotify.NewMailEventHandler(IHaveMail);
    }
    void IHaveMail(object sender, NewEmailEventArgs e)
    {
        Console.WriteLine("New Mail: {0}\n{1}",
                e.Subject, e.Message);
    }
    EmailNotify emailNotify;
}
class Test
{
   public static void Main()
   {
       EmailNotify emailNotify = new EmailNotify();
       MailWatch mailWatch = new MailWatch(emailNotify);
       emailNotify.NotifyMail("Hello!", "Welcome to Events!!!");
   }
}
```

The `EmailNotify` class now has a property named `NewMailEventHandler` rather than having an event of the same name. The property stores the delegate in a hash table rather than putting it in an event field, and then uses the static readonly object `onNewMailKey` to make sure that the property finds the proper delegate. Because object references are guaranteed by the system to be unique, creating a static readonly object is a nice way to generate a unique key at runtime.
In this situation, using a hash table is clearly a losing proposition, since it will take up more space than the previous version. This idiom is more valuable when there is a hierarchy of objects—such as controls that derive from the base class `Control` that have lots of sparse events. The `Control` class implements `GetEventHandler()` and `SetEventHandler()`, and all the controls that derive from it can use that to store their delegates.

Note that this is only a win if there are likely to be multiple instances of each control present at one time. If there aren't, the space used by the static readonly key will negate the space savings in the object.

Top

Prev

Chapter 24: User-Defined Conversions

Overview

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 are declared. This means that conversions can't be declared between two existing types, which makes the language simpler.

A Simple Example

This example implements a struct that handles roman numerals. It could also be written as a class.

```
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()
  {
        RomanNumeral numeral = new RomanNumeral(12);

        numeral = (RomanNumeral) 165;

        Console.WriteLine("Roman as int: {0}", (int)numeral);
        Console.WriteLine("Roman as string: {0}", (string)numeral);

        short s = numeral;
  }
}
```

This struct declares a constructor that can take a `short` value and it also declares a con- version 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. There is a conversion to `short` that is declared `implicit`, because the value in a `RomanNumeral` will always fit in a `short`. And finally, there's a conversion to `string` that gives the romanized version of the number. [1]

When an instance of this struct is created, the constructor can be used to set the value. An explicit conversion can be used to convert the integer value to a `RomanNumeral`. To get the romanized version of the `RomanNumeral`, the following would be written:

```
Console.WriteLine(roman);
```

If this is done, the compiler reports that there is an ambiguous conversion present. The class includes `implicit` conversions to both `short` and to `string`, and `Console.WriteLine()` has overloads that take both versions, so the compiler doesn't know which one to call.

In the example, an `explicit` cast is used to disambiguate, but it's a bit ugly. Since this struct would likely be used primarily to print out the romanized nota- tion, it probably makes sense to change the conversion to integer to be an `explicit` one so that the conversion to string is the only `implicit` one.

[1] No, this struct doesn't handle niceties such as replacing "IIII" with "IV", nor does it handle converting the romanized string to a `short`. The remainder of the implementation is left as an exercise for the reader.

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 that were declared in the struct itself. The user-defined conversions can also be used in situations where the source or destination types are not exact matches to the types in the conversion functions.

If the source or destination types are not exact matches, then the appropriate standard (i.e., 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 (or `explicit`) must also be compatible.

Perhaps an example will be a bit clearer. In the preceding example, the line

short s = numeral;

calls the implicit user-defined conversion directly. Since this is an implicit use of the user-defined conversion, there can also be another implicit conversion at the end:

int i = numeral;

Here, the implicit conversion from `RomanNumeral` to `short` is performed, followed by the implicit conversion from `short` to `long`.

In the explicit case, there was the following conversion in the example:

numeral = (RomanNumeral) 165;

Since the usage is `explicit`, the explicit conversion from `int` to `RomanNumeral` is used. Also, an additional 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 in a similar way, except that there are a few more situations to consider. Since the user conversion can be defined in either the source or destination type, there's a bit more design work to do, and the operation is a bit more complex. For details, see the "How It Works" section, later in this chapter.

Adding to the `RomanNumeral` example in the last section, a struct that handles binary numbers can be added:

```
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;
  }
}
```

The classes can be used together, but since they don't really know about each other, it takes a bit of extra typing. To convert from a `RomanNumeral` to a `BinaryNumeral` requires first converting to an `int`.

It would be nice to write the `Main()` function as

```
binary = roman;
roman = (RomanNumeral) binary;
```

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.

To get this, a user-defined conversion is required on either the `RomanNumeral` 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.

The classes are modified 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 also 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 considered.

For implicit conversions, it's pretty simple, and 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, the example will be modified to use classes rather than structs, and a new class that derives from `RomanNumeral` will be added:

```
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` is performed.

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.

The conversion could be rewritten 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. 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, the following guidelines should be considered.

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 system types, there is no option but to define the conversion as part of the class, since the source isn't available.
Even if the source *was* available, however, it would be really 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 there is no obvious simpler class. In that case, pick a class, and put both conversions there.

One Conversion to and from a Hierarchy

In my examples, there was 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 is good practice to do this and then to 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 is the most natural size for the type.

For example, in the `BinaryNumeral` class, there's an implicit conversion to `int`. If the user wants a smaller type, such as `short`, a cast can easily be done.

If there are multiple conversions 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, alternate versions of the con- versions should be supplied. If, for example, an object supports a conversion to `string`, it should also support calling `ToString()` on that function. Here's how it would be done 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, there are a few details on how the compiler views conversions that warrant a bit of explanation. Those who are really interested in the gory details can find them in the C# Language Reference. [2]

This section can be safely skipped.

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
{


}
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 only succeed if the `T` returned by the conversion is really only a `TBase`.

Revising things a bit, removing the conversion from `S` and adding it to `T`, we 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 class `T` isn't searched.

[2]The C# Language Reference can be found at http://msdn.microsoft.com/vstudio/nextgen/technology/csharpdownload.asp.

Chapter 25: Operator Overloading

Overview

OPERATOR OVERLOADING ALLOWS operators to be defined on a class or struct so that it can be used with operator syntax. This is most useful on data types where there is a good definition for what a specific operator means, thereby allowing an economy of expression for the user.

Overloading the relational operators `(==, !=, >, <, >=, <= )` is covered in the section that covers overloading the `Equals()` function from the .NET Frameworks, in Chapter 27, "Making Friends with the .NET Frameworks."

Overloading conversion operators is covered in Chapter 24, "User-Defined Conversions."

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. The following operators can be overloaded:

+ - ! ~ -- truefalse

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)

is not equivalent to

if (! (a == false))

This happens in SQL types, which have a null state that is 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 is no way to discriminate between the pre and post increment or decrement operation. Because the operators are static functions rather than member functions, this distinction is not 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 would typically return the type of the class or struct in which it is defined.

The following binary operators can be defined:

+ - * / % & | ^ << >> (relational 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 is not possible to overload member access, member invocation (function calling), or the +, `&&`, `||`, `?:`, or `new` operators. This is for the sake of simplicity; while one can do interesting things with such overloadings, it greatly increases the difficulty in understanding code, since programmers would have to always remember that member invocation (for example) could be doing something special. [1]`New` 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 is also not possible to overload the compound assignment operators `+=`, `*=`, etc., since they are always expanded to the simple operation and an assignment. This avoids cases where one would be defined and the other wouldn't be, or (shudder) they would be defined with different meanings.

[1]One could, however, argue that member access can be overloaded through properties.

Design Guidelines

Operator overloading is a feature that should be used only when necessary. By "necessary," I mean that it makes things easier and simpler for the user.

Good examples of operator overloading would be defining arithmetic operations on a complex number or matrix class.
Bad examples would be 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, it shouldn't be defined as an operator. Don't make up new meanings for operators.
In practice, the equality (==) and inequality (`!=`) operators are the ones that will be defined most often, since if this is not done, there may be 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, value types can't be used as base classes, and `Int32` is a value type. Second, even if it was possible, it wouldn't really work for `BinaryNumeral`, because a `BinaryNumeral` isn't an integer; it only supports a small part of the possible integer range. Because of this, derivation would not be a good design choice. The smaller range means that even if `BinaryNumeral` as derived from `int`, there isn't an implicit conversion from `int` to `BinaryNumeral`, and any expressions would therefore require casts.

Even if these 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.

[2]As we saw earlier, if your type is a reference type (class), using == will compare to see 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, which may be sufficient.

Chapter 26: Other Language Details

Overview

THIS CHAPTER DEALS WITH 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 from other examples:

```
using System;
class Test
{
    public static void Main()
    {
        Console.WriteLine("Hello, Universe!");
    }
}
```

Returning an Int Status

It will often be useful to return a status from the `Main()` function. This is done 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

The command-line parameters to an application can be accessed by declaring the `Main()` function with a `string` array as a parameter. The parameters can then be processed 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 Mains

It is 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#, this `static` test function can be written as a `Main()` function, which makes automating such tests easy.
If there is a single `Main()` function encountered during a compilation, the C# compiler will use it. If there is more than one `Main()` function, the class that contains the desired `Main()` can be specified on the command line with the `/main:<classname>` option.

```
// error
using System;
class Complex
{
   public 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.

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 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 are used to control the compilation of code.

Getting rid of the macro version of `#define` allows the programmer to understand more clearly what the program is saying. A name that isn't familiar must come from one of the namespaces, and there's no need to hunt through `include` files to find it.

One of the biggest reasons for this change is that getting rid of preprocessing and `#include` enables a simplified compilation structure, and therefore we get some impressive improvements in compilation speed.[1] Additionally, there is no need to write a separate header file and keep it in sync with the implementation file.

When C# source files are compiled, the order of the compilation of the individual files is unimportant, [2] and it is equivalent to them all being in one big file. There is no need for forward declarations or worrying about the order of `#includes`.

[1]When I first installed a copy of the compiler on my system, I typed in a simple example and compiled it, and it came back fast—so fast that I was convinced that something was wrong, and hunted down a developer for assistance. It's so much faster than C++ compilers are (or can be).

[2]Except for the fact that the output file will automatically use the name of the first compiland.

Preprocessing Directives

The following preprocessing directives are supported:

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	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.

Here's an example of how they might be used:

```
#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, or an error occurs. The previous example can't be written 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
    }
}
```

Preprocessor Expressions

The following operators can be used in preprocessor expressions:

OPERATOR	DESCRIPTION
`! ex`	Expression is true if `ex` is false
`ex == value`	Expression is true if `ex` is equal to `value`
`ex != value`	Expression is true if `ex` is not equal to `value`
`ex1 && ex2`	Expression is true if both `ex1` and `ex2` are true
`ex1 || ex2`	Expression is true if either `ex1` or `ex2` are true

Parentheses can be used 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, there are a few other preprocessor functions that can be used.

#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, the following could be done:

#warning Check algorithm with John

This would result in the string “Check algorithm with John” being output when the line was compiled.

#line

With `#line`, the programmer can specify the name of the source file and the line number that are reported when the compiler encounters errors. This would typically be used with machine-generated source code, so the reported lines can be synced with a different naming or numbering system.

Lexical Details

The lexical details of the language deal with things 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 is 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] Unicode characters can be specified 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, an "at" character `(@)` can be placed before the name, which merely indicates to C# that the name is not a keyword, but an identifier.
Similarly, use "`@`" to use 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 is not recommended because of the confusion it can create.

Keywords

Keywords are reserved words that cannot be used as identifiers. The keywords in C# are:

Literal

abstract	base	bool	break	by e
case	catch	char	checked	class
cons	continue	decimal	default	delegate
do	double	else	enum	even
explicit	extern	false	finally	fixed
float	for	foreach	goto	if
implicit	in	int	interface	internal
is	lock	long	namespace	new
null	object	operator	out	override
params	private	protected	public	readonly
ref	return	sbyte	sealed	short
sizeof	static	string	struct	switch
this	throw	rue	try	typeof
uint	ulong	unchecked	unsafe	ushort
using	virtual	void	while	

Literals are the way in which values are written for variables.

Boolean
There are two Boolean literals: `true` and `false`.

Integer
Integer literals are written simply by writing the numeric value. Integer literals that are small enough to fit into the `in` data type[4] are treated as `int`s ; if they are too big to fit into an `in`, they will be created as the smallest type of `uint, long,` or `ulong` in which the literal will fit.

Some integer literal examples:

123

-15

Integer literals can also be written in hexadecimal format, by placing “0x” in front of the constant:

0xFFFF

0x12AB

Real
Real literals are used for the types `float, double`, and `decimal`. Float literals have "`f`" or "`F`" after them; double literals have "`d`" or "`D`" after them, and `decimal` literals have "`m`" or "`M`" after them. Real literals without a type character are interpreted as `double` literals.
Exponential notation can be used by appending “`e`" followed by the exponent to the real literal.

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 following escape sequences are supported:

ESCAPE SEQUENCE	DESCRIPTION
`\'`	Single quote
`\"`	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.

String

String literals are written as a sequence of characters enclosed in double quotes, such as "`Hello`". All of the character escape sequences are supported within strings.

Strings cannot span multiple lines, but the same effect can be achieved by concatenating them together:

```
string s = "What is your favorite color?" +
        "Blue. No, Red. ";
```

When this code is compiled, a single string constant will be created, consisting of the two strings concatenated together.

Verbatim Strings

Verbatim strings allow some strings to be specified more simply.

If a string contains the backslash character, such as a filename, a verbatim string can be used to turn off the support for escape sequences. Instead of writing something like

```
string s = "c:\\Program Files\\Microsoft Office\\Office";
```

the following can be written:

```
string s = @"c:\Program Files\Microsoft Office\Office";
```

The verbatim string syntax is also useful if the code is generated by a program and there is no way to constrain the contents of the string. All characters can be represented within such a string, though any occurrence of the double-quote character must be doubled:

```
string s = @"She said, ""Hello""";
```

In addition, strings that are written with the verbatim string syntax can span multiple lines, and any whitespace (spaces, tabs, and newlines) is preserved.

```
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 `/*` and `*/` to denote 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 is used to associate documentation with code; those comments are described in the XML documentation section of Chapter 31, “Deeper into C#.”

[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 characters other than the underscore (_) can also be used for combinations. See the C# Language Reference (http://msdn.microsoft.com/vstudio/nextgen/technology/csharpdownload.asp) for a full description.

[4] See the "Basic Data Types" section in Chapter 3, "C# Quickstart."

Chapter 27: Making Friends with the .NET Frameworks

Overview

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 framework. This chapter will detail how to make user-defined objects operate more like the objects in the .NET Runtime and Frameworks.

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 Frameworks 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 will result in the following:

```
Employee: Employee
```

By overloading `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;
   }
     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 us 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 more control over formatting is desired, such as implementing a floating point class with different formats, the `IFormattabl` interface can be overridden. `IFormattable` is covered in the "Custom Object Formatting" section of Chapter 30, ".NET Frameworks Overview."

Equals()

`Equals()` is used to determine 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:

```
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)
   {
```

```
        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));
    }
    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 == h rbClone);
    }
}
```

This will produce 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 cannot be overloaded separately. [1]

[1] This is required for two reasons. The first is that if a user uses ==, they can expect != to work as well. The other is to support nullable types, for which a == b does *not* imply !(a != b).

Hashes and `GetHashCode()`

The Frameworks include the **`Hashtable`** class, which is very useful for doing fast lookup 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.
The default implementation of **`GetHashCode()`** doesn't work this way, and there- fore it must be overridden to work correctly. If not overridden, the hash code will only be identical for the same instance of an object, and a search for an object that is equal but not the same instance will fail.

If there is 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 is used for the hash code. In the **`Main()`** function, several employees are created, and they are 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 adds the hash codes of the elements together, and returns them.

[2] I've always liked Robert Sedgewick's *Algorithms in C* as a good introduction.

[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.

Chapter 28: System.Array and the Collection Classes

Overview

CONCEPTUALLY, THIS CHAPTER will give an overview of what classes are available. It will then cover them by class and give examples of what interfaces and functions are required to enable specific functionality.

Sorting and Searching

The Frameworks collection classes provide some useful support for sorting and searching, with built-in functions to do sorting and binary searching. The `Array` class provides the same functionality but as static functions rather than 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:

```
Element 1
Element 4
Element 5
Element 10
Element 33
Element 100
```

This is very 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 Frameworks have some very 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);
    }
}
```

This program gives us the following output:

```
Employee: George:1
Employee: Fred:2
Employee: Bob:3
Employee: Tom:4
```

This implementation only allows one sort ordering; the class could be defined to sort based on employee ID or based on name, but there's no way to allow the user to choose which sort order they prefer.

Using `IComparer`

The designers of the Frameworks 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 only implement an interface once, which would allow only a single sort order.[1] A separate class is needed for each sort order, with the class implementing `IComparer`. The class will be very simple, since all it will do is implement the `Compare()` function:

```
using System;
```

```
using System.Collections;
class Employee
{
   public string name;
}
class SortByNameClass: IComparer
{
   public int Compare(object obj1, object obj2)
   {
      Employee emp1 = (Employee) obj1;
      Employee emp2 = (Employee) obj2;
      return(String.Compare(emp1.name, mp2.name));
   }
}
```

`The Compare()` member takes two objects as parameters. Since the class should only be used for sorting employees, the `object` parameters are cast to `Employee`. The `Compare()` function built into `string` is then used for the comparison.

The `Employee` class is then revised as follows. The sort-ordering classes are placed 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`. This can be simplified 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 `SortByNam` 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 `SortByNam` and `SortById` properties return an `IComparer`, so it's obvious that 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 is 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:

```
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}", g orge < fred);
        Console.WriteLine("Tom >= Bob: {0}", tom >= bob);
    }
}
```

This example produces the following output:

```
George < Fred: false
Tom >= Bob: true
```

Advanced Use of Hashes

In some situations, it may be desirable to define more than one hash code for a specific object. This could be used, for example, to allow an Employee to be searched for based on the employee ID or on the employee name. This is done 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`:

```
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}", h rbClone, address);
    }
}
```

This technique should be used 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.

ICloneable

The `object.MemberWiseClone()` function can be used 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 ContainedValu
{
    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 `Contained-Value` object affects both instances of `MyObject`.

What is needed is a deep copy, where a new instance of `ContainedValue` is created for the new instance of `MyObject`. This is done by implementing the `ICloneable` interface:

```
using System;
class ContainedValu
{
   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 the contents of this instance can be modified without affecting the contents of `my`. Unlike some of the other interfaces that might be defined on an object, `IClonable` is not called by the runtime; it is provided merely to ensure that the `Clone()` function has the proper signature.

[1] `IComparable` *could* implement one sort order and `IComparer` another, but that would be very confusing to the user.

Design Guidelines

The intended use of an object should be considered when deciding which virtual functions and interfaces to implement. The following table provides guidelines for this:

OBJECT USE	FUNCTION OR INTERFACE
General	`ToString()`
Arrays or collections	`Equals()`, `operator==()`, `operator!=()`, `GetHashCode()`
Sorting or binary search	`IComparable`
Multiple sort orders	`IComparer`
Multiple has lookups	`IHashCodeProvider`

Functions and Interfaces by Framework Class

The following tables summarize which functions or interfaces on an object are used by each collection class.

Array

FUNCTION	USES
`IndexOf()`	`Equals()`
`LastIndexOf()`	`Equals()`
`Contains()`	`Equals()`
`Sort()`	`Equals()`, `IComparable`
`BinarySearch()`	`Equals()`, `IComparable`

ArrayList

FUNCTION	USES
`IndexOf()`	`Equals()`
`LastIndexOf()`	`Equals()`
`Contains()`	`Equals()`
`Sort()`	`Equals()`, `IComparable`
`BinarySearch()`	`Equals()`, `IComparable`

Hashtable

FUNCTION	USES
`HashTable()`	`IHashCodeProvider`, `IComparabl` (optional)
`Contains()`	`GetHashCode()`, `Equals()`
`Item`	`GetHashCode()`, `Equals()`

SortedList

FUNCTION	USES
`SortedList()`	`IComparable`
`Contains()`	`IComparable`
`ContainsKey()`	`IComparable`
`ContainsValue()`	`Equals()`
`IndexOfKey()`	`IComparable`
`IndexOfValue()`	`Equals()`
`Item`	`IComparable`

Chapter 29: Interop

Overview

ONE OF THE IMPORTANT CAPABILITIES of C# is being able to interoperate with existing code, whether it be COM-based or in a native DLL. This chapter provides a brief overview of how interop works.

Using COM Objects

To call a COM object, the first step is to define a proxy (or wrapper) class that defines the functions in the COM object, along with additional information. This is a fair amount of tedious work, which can be avoided in most cases by using the `tlbimp` utility. This utility reads the COM typelib information and then creates the proxy class. This will work in many situations, but if more control is needed over marshalling, the proxy class may need to be written by hand. In this case, attributes are used to specify how marshalling should be performed.

Once the proxy class is written, it is used like any other .NET class, and the runtime handles the ugly stuff.

Being Used by COM Objects

The runtime also allows .NET objects to be used in place of COM objects. The `tlbexp` utility is used to create a typelib that describes the COM objects, so that other COM-based programs can determine the object's interface, and the `regasm` utility is used to register an assembly so that it can be accessed through COM. When COM accesses a .NET class, the runtime handles creating 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 functions written in native code through a runtime feature known as "plat- form invoke." The file that the function is located in is specified by the `sysimport` attribute, which can also be used to specify the default character marshalling. In many cases, that attribute is all that will be needed, but if a value is passed by reference, the `in` and `out` attributes may be specified to tell the marshaller how to pass the value.

```
class Test
{
   [sysimport(dll="user32.dll")]
   public static extern int MessageBoxA(int h, string m,
      string c, int type);
   public static void Main()
   {
      int retval = MessageBoxA(0, "Hello", "Caption", 0);
   }
}
```

When this code runs, a message box will appear.

Chapter 30: .NET Frameworks Overview

Overview

THE .NET FRAMEWORKS CONTAIN many functions that are normally found in language- specific runtime libraries, and it is therefore important to understand what classes are available in the Frameworks.

Numeric Formatting

Numeric types are formatted through the `Format()` member function of that data type. This can be called directly, through `String.Format()`, which calls the `Format()` function of each data type, or `Console.WriteLine()`, which calls `String.Format()`.

Adding formatting to a user-defined object is discussed in the "Custom Object Formatting" section, later in this chapter. This section discusses how formatting is done with the built-in types.

There are two methods of specifying numeric formatting. A standard format string can be used to convert a numeric type to a specific string representation. If further control over the output is desired, a custom format string can be used.

Standard Format Strings

A standard format string consists of a character specifying the format, followed by a sequence of digits specifying the precision. The following formats are supported:

FORMAT CHARACTER	DESCRIPTION
C, c	Currency
D, d	Decimal
E, e	Scientific (exponential)
F, f	Fixed-point
G, g	General
N, n	Number
X, x	Hexadecimal

Currency

The currency format string converts the numerical value to a string containing a locale-specific currency amount. By default, the format information is determined by the current locale, but this may be changed by passing a `NumberFormatInfo` object.

```
using System;


class Test
{
   public static void Main()
   {
      Console.WriteLine("{0:C}", 33345.8977);
      Console.WriteLine("{0:C}", -33345.8977);
   }
}
```

This example gives the following output:

```
$33,345.90
($33,345.90)
```

Decimal

The decimal format string converts the numerical value to an integer. The minimum number of digits is determined by the precision specifier. The result is left-padded with zeroes to obtain the required number of digits.

```
using System;
class Test
{
   public static void Main()
```

```
   {
      Console.WriteLine("{0:D}", 33345);
      Console.WriteLine("{0:D7}", 33345);
   }
}
```

This example gives the following output:

```
33345
0033345
```

Scientific (Exponential)

The scientific (exponential) format string converts the value to a string in the form

m.dddE+xxx
One digit always precedes the decimal point, and the number of decimal places is specified by the precision specifier, with six places used as the default. The format specifier controls whether "E" or "e" appears in the output.

```
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);
   }
}
```

This example 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.

```
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);
   }
}
```

This example gives the following output:

```
33345.90
```

33346

33345.89770

General

The general format string converts the value to either a fixed-point or scientific format, whichever one gives a more compact format.

```
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);
   }
}
```

This example 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, such as

12,345.11

The format may be controlled by passing a `NumberFormatInfo` object to the `Format()` function.

```
using System;


class Test
{
   public static void Main()
   {
      Console.WriteLine("{0:N}", 33345.8977);
      Console.WriteLine("{0:N4}", 33345.8977);
   }
}
```

This example gives the following output:

33,345.90

33,345.8977

Hexadecimal

The hexadecimal format string converts the value to hexadecimal format. The minimum number of digits is set by 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.

```
using System;
```

```
class Test
{
   public static void Main()
   {
      Console.WriteLine("{0:X}", 255);
      Console.WriteLine("{0:x8}", 1456);
   }
}
```

This example gives the following output:

```
FF
000005b0
```

NumberFormatInfo

The `NumberFormatInfo` class is used to control the formatting of numbers. By setting the properties in this class, the programmer can control the currency symbol, decimal separator, and other formatting properties.

Custom Format Strings

Custom format strings are used to obtain more control over the conversion than is available through the standard format strings. In custom format strings, special characters form a template that the number is formatted into. Any characters that do not have a special meaning in the format string are copied verbatim to the output.

Digit or Zero Placeholder

The zero (`0`) character is used as a digit or 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.

```
using System;
class Test
{
   public static void Main()
   {
      Console.WriteLine("{0:000}", 55);
      Console.WriteLine("{0:000}", 1456);
   }
}
```

This example gives the following output:

```
055
1456
```

Digit or Space Placeholder

The pound (`#`) character is used as the digit or space placeholder. It works exactly the same as the "`0`" placeholder, except that a blank appears if there is no digit in that position.

```
using System;


class Test
{
   public static void Main()
```

```
    {
        Console.WriteLine("{0:#####}", 255);
        Console.WriteLine("{0:#####}", 1456);
        Console.WriteLine("{0:###}", 32767);
    }
}
```

This example 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. The character used as the decimal separator in the formatted string is controlled by a `NumberFormatInfo` instance.

```
using System;
class Test
{
    public static void Main()
    {
        Console.WriteLine("{0:#####.000}", 75928.3);
        Console.WriteLine("{0:##.000}", 1456.456456);
    }
}
```

This example gives the following output:

```
75928.300
1456.456
```

Group Separator

The comma (`,`) character is used as a group separator. If a "`,`" 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. The character used in the formatted string and the number of numbers to group together is controlled by a `NumberFormatInfo` instance.

```
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);
    }
}
```

This example gives the following output:

```
2,555,634,323
```

14,563,553.594

14,563,553.594

Number Prescaler

The comma (,) character can also be used to indicate that the number should be prescaled. In this usage, the "," must come directly before the decimal point or at the end of the format string.
For each "," that is present in this location, the number is divided by 1,000 before it is formatted.

```
using System;
class Test
{
   public static void Main()
   {
      Console.WriteLine("{0:000,.##}", 158847);
      Console.WriteLine("{0:000,,,.###}", 1593833);
   }
}
```

This example gives the following output:

158.85

000.002

Percent Notation

The percent (%) character is used to indicate that the number to be displayed should be displayed as a percentage. The number is multiplied by 100 before it is formatted.

```
using System;
class Test
{
  public static void Main()
  {
     Console.WriteLine("{0:##.000%}", 0.89144);
     Console.WriteLine("{0:00%}", 0.01285);
   }
}
```

This example 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 "+" means that there will be a plus or minus in that position, while a "-" means there is a character there only if the number is negative.

```
using System;
class Test
{
   public static void Main()
   {
      Console.WriteLine("{0:###.000E-00}", 3 1415533E+04);
```

```
        Console.WriteLine("{0:#.0000000E+000}", 2.553939939E+101);
    }
}
```

This example gives the following output:

```
314.155E-02
2.5539399E+101
```

Section Separator

The semicolon (;) character is used to specify different format strings for a number, depending on whether the number is positive, zero, or negative. 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, the zero value, and negative values.

```
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);
    }
}
```

This example gives the following output:

```
457
(.00)
456.55
```

Escapes and Literals

The slash (\) character can be used to escape characters so that they aren't interpreted as formatting characters. Because the "\" character already has meaning within C# literals, it will be easier to specify the string using the verbatim literal syntax; otherwise, a "\\" is required to generate a single "\" in the output string.
A string of uninterpreted characters can be specified by enclosing them in single quotes; this may be more convenient than using " ".

```
using System;
class Test
{
    public static void Main()
    {
        Console.WriteLine("{0:###\\#}", 255);
        Console.WriteLine(@"{0:###\#}", 255);
        Console.WriteLine("{0:###'#0%;'}", 1456);
    }
}
```

This example gives the following output:

```
255#
255#
```

1456#0%;

Date and Time Formatting

The `DateTime` class provides flexible formatting options. Several single-character formats can be specified, and custom formatting is also supported.

Standard `DateTime` Formats

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	UniversalSortableDateTime-Pattern

Custom DateTime Format

The following patterns can be used to build a custom format:

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, 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 additional information with a different format string.

```
using System;
class Employee
{
  public Employee(int id, string firstName, string lastName)
  {
    this.id = id;
    this.firstName = firstName;
    this.lastName = lastName;
  }
public string Format (string format, IServiceObjectProvider sop)
{
    if (format.Equals("F"))
       return(String.Format("{0}: {1}, {2}",
          id, lastName, firstName));
```

```
    else
        return(id.Format(format, sop));
  }
  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);
  }
}
```

The `Format()` function looks for the "`F`" format. If it finds it, it writes out 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.

New Formatting for Existing Types

It is also possible to supply a new format for existing objects. The following object will allow `float` and `double` values to be formatted in angstroms. An angstrom is equal to 1E-10 meters.

```
using System;


public class AngstromFormatter: IServiceObjectProvider, ICustomFormatter
{
  public object GetServiceObject(Type service)
  {
     if (service == typeof(ICustomFormatter))
        return this;
     else
        return null;
  }


  public string Format(string format, object arg,
    IServiceObjectProvider sop)
  {
    if (format == null)
       return(String.Format("{0}", arg));


    if (format.StartsWith("Ang"))
    {
          // extract any extra formatting information
          // after "Ang" here
       string extra = "";
```

```
        if (arg is float)
        {
           float f = (float) arg;
           f *= 1.0E10F;
           return(String.Format("{0:" + extra + "}", f) + " Å");
        }
        else if (arg is double)
        {
           double d = (double) arg;
           d *= 1.0E10D;
           return(String.Format("{0:" + extra + "}", d) + " Å");
        }
     }
        // not an object or format we support
     return(String.Format("{0:" + format + "}", arg));
  }
}
class Test
{
  public static void Main()
  {
     AngstromFormatter angstrom = new AngstromFormatter();

     Console.WriteLine("Meters: {0}", 1.35E-8F, angstrom);
     Console.WriteLine(String.Format("Angstroms: {0:Ang}",
        new object[] {1.35E-8F}, angstrom));
     Console.WriteLine(String.Format("Angstroms: {0:Ang:g}",
        new object[] {3.59393E-9D}, angstrom));
  }
}
```

In this example, the `AngstromFormatter` class supports formatting numbers in angstroms by dividing the values by 1E-10, and then appending the angstrom symbol "Å" to the string. Further formatting may be specified after the "`Ang`" format to control the appearance of the floating point number.

Numeric Parsing

Numbers are parsed using the `Parse()` method provided by the numeric data types. Flags from the `NumberStyles` class can be passed to specify which styles are allowed, and a `NumberFormatInfo` instance can be passed 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.

```
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);
    }
}
```

This example produces the following output.

99953

1. 3433E35

Using XML in C#

While C# does support XML documentation (see the "XML Documentation" section in Chapter 31, "Deeper into C#"), C# doesn't provide any language support for using XML.
That's okay, however, because the Common Language Runtime provides extensive support for XML. Some areas of interest are the `System.Data.Xml` and `System.Xml` namespaces.

Input/Output

The .NET Common Language Runtime 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, etc.
Reading and writing is done 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 following classes are available:

I/O Classes derived from `Stream`

CLASS	DESCRIPTION
`FileStream`	A stream on a disk file
`MemoryStream`	A stream that is stored in memory
`NetworkStream`	A stream on a network connection
`BufferedStream`	Implements a buffer on top of another stream

With the exception of `BufferedStream`, which sits 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 there are several supplied ones that can be selected depending on what final format is desired.

Binary

The `BinaryReader` and `BinaryWriter` classes are used to read and write values in binary (or raw) format. For example, a `BinaryWriter` can be used to write an `int`, followed by a `float`, followed by another `int`.

These classes are typically used—not surprisingly—to read and write binary formats. They 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, etc. There are two different implementations of `TextReader` available.
The somewhat strangely named `StreamWriter` class is the one used for "normal" I/O (open a file, read the lines out), and operates on a `Stream`.
The `StringReader` and `StringWriter` classes can be used to read and write from a string.

XML

The `XmlTextReader` and `XmlTextWriter` classes are used to read and write XML. They are similar to `TextReader` and `TextWriter` in design, but they do not derive from those classes because they deal with XML entities rather than text. They are low- level classes that are used to create or decode XML from scratch.

Reading and Writing Files

There are 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();
   }
}
```

It is also possible to use the functions in the `File` class to get a stream to a file. This is most useful if there is already a `File` object with the file information available, as in the `PrintFile()` function in the next example.

Traversing Directories

This example shows how to traverse a directory structure. It defines 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(Directory dir, int level, object obj);
  public delegate void ProcessFileCallback(File 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 Directory(rootDir), 0, obj);
  }
  void DoWalk(Directory dir, int level, object obj)
  {
    foreach (FileSystemEntry d in dir.GetFileSystemEntries ())
    {
      if (d is File)
      {
        if (fileCallback != null)
          fileCallback((File) d, level, obj);
      }
      else
      {
        if (dirCallback != null)
          dirCallback((Directory) d, level, obj);
        DoWalk((Directory) d, level + 1, obj);
      }
    }
  }



  ProcessDirCallback dirCallback;
  ProcessFileCallback fileCallback;
}

class Test
{
  public static void PrintDir(Directory d, int level, object obj)
  {
    WriteSpaces(level * 2);
    Console.WriteLine("Dir: {0}", d.FullName);
  }
  public static void PrintFile(File 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");
    }
}
```

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.
This help is provided through two attributes. The `[Serializable]` attribute is used to mark an object as okay to serialize. The `[NonSerialized]` attribute can be applied to a field or property to indicate that it shouldn't be serialized. This is useful if it is 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 the 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 /r:system.runtime.serialization.formatters.soap.dll
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` is not 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 a SOAP encoding. To produce a specific XML encoding, use the `XmlSerializer` class.

Threading

The `System Threading` namespace contains classes useful for threading and synchronization. The appropriate type of synchronization and/or exclusion depends upon the design of the program, but C# supports simple exclusion using the `lock` statement.
`Lock` uses the `System Threading.Monitor` class and provides similar functionality to the `CriticalSection` calls in Win32.

The following example simulates incrementing an account balance. The code that increments the balance first fetches the current balance into a temporary variable, and then sleeps for millisecond. During this sleep period, it's very likely that another thread will fetch the balance before the first thread can wake up and save the new value.
When run as written, the final balance will be much less than the expected value of 1,000. By removing the comments on the `lock` statement in the `Deposit()` function, the system will ensure that only one thread can be in the `lock` block at a time, and the final balance will be correct.
The object passed to the `lock` statement must be a reference type, and it should contain the value that is being protected. Locking on the current instance with `this` will protect against any access by the same instance.

```
using System;
using System.Threading;


public class Account
{
   public Account(decimal balance)
   {
      this.balance = balance;
   }

   public void Deposit(decimal amount)
   {
      //lock(this) // uncomment to protect block
      {
         Decimal temp = balance;
         temp += amount;
         Thread.Sleep(1); // deliberately wait
         balance = temp;
      }
   }
   public Decimal Balance
   {
      get
      {
         return(balance);
      }
   }
   decimal balance;
}


class ThreadTest
{
```

```
public void MakeDeposit()
{
    for (int i = 0; i < 10; i++)
        account.Deposit(10);
}    public static void Main(string[] args)
{
    ThreadTest b = new ThreadTest();
    Thread t = null;
        // create 10 threads.
    for (int threads = 0; threads < 10; threads++)
    {
        t = new Thread(new ThreadStart(b.MakeDeposit));
        t.Start();
    }
    t.Join(); // wait for last thread to finish
    Console.WriteLine("Balance: {0}", b.account.Balance);
}
Account account = new Account(0);
```

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 Microsoft’s Money Central site, and then extract the quote out of the HTML page using a regular expression.

```
// file: quote.cs
// compile with: csc quote.cs /r:system.net.dll /
r:system.text.regularexpressions.dll
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 = WebRequestFactory.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).Group(1).ToString();
    }
    string symbol;
    string last;
}

    class Test
    {
      public static void Main(string[] args)
      {
        if (args.Length != 1)
            Console.WriteLine("Quote <symbol>");
        else
        {
            QuoteFetch q = new QuoteFetch(args[0]);
            Console.WriteLine("{0} = {1}", args[0], q Last);
        }
      }
}
```

Chapter 31: Deeper into C#

Overview

THIS CHAPTER WILL DELVE 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 there are some guidelines from the .NET Common Language Runtime that should be considered.

These guidelines are detailed in "Class Library Design Guidelines" in the .NET documentation and 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 Common Language Runtime classes and samples also have many examples.

Naming

There are two naming conventions that are used.

- PascalCasing capitalizes the first character of the first word.
- camelCasing is the same as PascalCasing, except the first character of the first word isn't capitalized.

In general, PascalCasing is used for anything that would be visible externally from a class, such as classes, enums, methods, etc. The exception to this is method parameters, which are defined using camelCasing.

Private members of classes, such as fields, are defined using camelCasing.

There are a few other conventions in naming:

- Avoid common keywords in naming, to decrease the chance of collisions in other languages.
- Event classes should end with `EventArgs`.
- Exception classes should end with `Exception`.
- Interfaces should start with `I`.
- Attribute classes should end in `Attribute`.

Encapsulation

In general, classes should be heavily encapsulated. 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 change.

Guidelines for the Library Author

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 Common Language Specification. This specification details what features a language should support to be a .NET-compliant language, and can be found 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, there are the following restrictions:

- Unsigned types can't be exposed as part of the public interface of a class. They can be freely used 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, they can be used 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, namespaces should be named using the `CompanyName.TechnologyName` convention. For example, the full name of a class to control an X-ray laser would be something like:

AppliedEnergy.XRayLaser.Controller

Unsafe Code

There are many benefits of code verification in the .NET runtime. Being able to verify that code is type-safe not only enables download scenarios, 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, more control is needed. In these situations, unsafe code can be used.

Unsafe means that the runtime cannot verify that the code is safe to execute. It therefore can only be executed if the assembly has full trust, which means it cannot 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.
There are three versions of the function that clones arrays of points. `Clone-PointArray()` is written without using unsafe features, and merely copies the array entries over. The second version, `ClonePointArrayUnsafe()`, uses pointers to iterate through the memory and copy it over. The final version, `ClonePointArrayMemcpy()`, calls the system function `CopyMemory()` to perform the copy.

To give some time comparisons, the following code is instrumented.

```
using System;
using System.Diagnostics;


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
[sysimport(dll = "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 that it can use any of the clone functions. It uses the `Counter` class, which provides access to the system timers. The frequency—and accuracy—of this class will vary based upon the version of Windows that is being used.

As with any benchmarking, the initial state of memory is very important. To help control for this, `TimeFunction()` does each method 5 times and only prints out the shortest time. Typically, the first iteration is slower, because the CPU cache isn't ready yet, and subsequent times get faster. For those interested, these times were generated on a 500 MHz Pentium running Windows 2000 Professional, but they were generated with pre-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`. The results are summarized below:

METHOD	P=10, I=2,000,000	P=1,000, I=20,000	P=100,000, I=200
Baseline	1.562	0.963	3.459
Unsafe	1.486	1.111	3.441
Memcpy	2.028	1.121	2.703

For small arrays, the unsafe code is fastest, and for very 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 here is that the unsafe code isn't a clear win over the baseline code.

The lesson in all this is that unsafe code doesn't automatically mean faster code, and that it's important to benchmark when doing performance work.

Structure Layout

The runtime allows a structure to specify the layout of its data members, using the `StructLayout` attribute. By default, the layout of a structure is automatic, which means that the runtime is free to rearrange the fields. When using interop to call into native or COM code, better control may be required.

When specifying the `StructLayout` attribute, three kinds of layout can be specified using the `LayoutKind` enum:

- Sequential, where all fields are in declaration order. For sequential layout, the Pack property can be used 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.
- `Union`, where all members are assigned offset 0.

Additionally, the `CharSet` property can be specified to set the default marshalling for string data members.

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 in some information that only a compiler can get consistently correct, and write it out to a separate file.

C# XML support can be divided into two sections: compiler support and documentation convention. In the compiler support section, there are tags that 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 are processed using information that is only known to the compiler. The following example illustrates the use of the support tags:

```
// file: employee.cs
using System;
namespace Payroll
{

/// <summary>
/// The Employee class holds data about an employee.
/// This class 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 that 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,

```
<see cref="String">
```

becomes

```
<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:

```
<seealso cref="Employee(int, string)">
```

becomes

```
<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 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; an XSL file that specifies how the XML should be rendered can be added, which would lead to the display shown in Figure 31-1 in a browser that supports XSL.

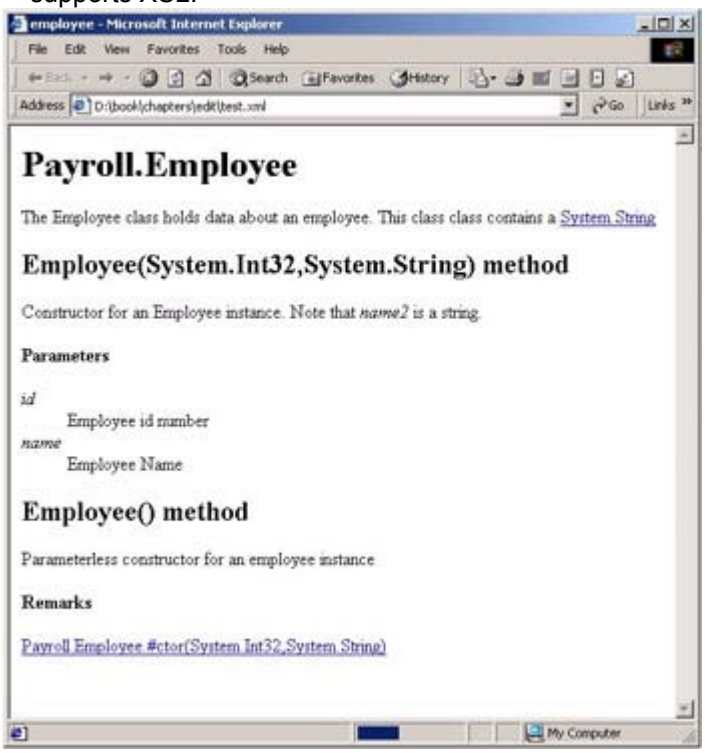


Figure 31-1. XML file in Internet Explorer with formatting specified by an XSL file

XML Documentation Tags

The remainder of the XML documentation tags describe the .NET documentation convention. They can be extended, modified, or ignored if necessary for a specific project.

TAG	DESCRIPTION
`<Summary>`	A short description of the item

<Remarks>	A long description of an item
<c>	Format 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

Garbage Collection in the .NET Runtime

Garbage collection has a bad reputation in a few areas of the software world. Some programmers feel that 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.

This section will 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 here describes the situation for platforms such as the PC. Systems with more constrained resources are likely to have simpler GC systems.

Note also that there are optimizations performed for multiproc and server machines.

Allocation

Heap allocation in the .NET Runtime world is very fast; all the system has to do is make sure that there's enough room in the managed heap 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 that there's enough room, the system might have to perform a garbage collection.

To improve performance, large objects (>20K) 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 that are 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, there can't be any other operations going on in the system. In other words, all useful work must be stopped 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 around. 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, and 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 31-2 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, though there is space for additional objects to be allocated.

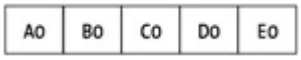


Figure 31-2. Initial memory state before any garbage collection

At the time of the first garbage collection, `B` and `D` are the only objects that are still in use. The heap looks like Figure 31-3 after collection.

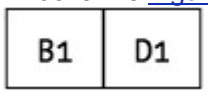


Figure 31-3. Memory state after first garbage collection

Since `B` and `D` survived a collection, their generation is incremented to 1. New objects are then allocated, as shown in Figure 31-4.

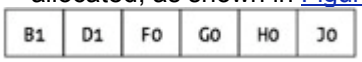


Figure 31-4. New objects are allocated

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 31-5.

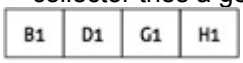


Figure 31-5. Memory state after a generation 0 collection

Even though `B` is no longer live, it doesn't get collected because the collection was only for generation 0. After a few new objects are allocated, the heap looks like Figure 31-6.

Figure 31-6. 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, and leads to the layout shown in Figure 31-7.

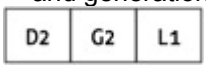


Figure 31-7. Memory state after a generation 0 and generation 1 garbage collection

Finalization

The garbage collector supports a concept known as finalization, which is somewhat analogous to destructors in C++. In C#, they are known as destructors and are declared with the same syntax as C++ destructors, but from the runtime perspective, they are 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 discussing their limitations, it's useful to understand 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 is 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 is 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.
- There is no guaranteed order for finalization. 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.
- Finalizers aren't called on normal program exit, to speed up exit. This can be controlled, but is discouraged.

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. This should be done 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. This is useful for the times where the behavior of a program won't be obvious to the runtime. If, for example, the program has just finished a bunch of processing and is getting rid of a considerable number of objects, it might make sense to do a collection at that point.

Forcing Finalization on Exit

If it's really important to have all finalizers called on exit, the `System.GC.RequestFinalizeOnShutdown()` method can be used. This may slow down the shutdown of the application.

Suppressing Finalization

As mentioned earlier, an instance of an object is placed on the finalization list when it is created. If it turns out that an object doesn't need to be finalized (because the cleanup function has been called, for example), the `System.GC.SupressFinalize()` function can be used to remove the object from the finalization list.

Deeper Reflection

Examples in the attributes section showed how to use reflection to determine the attributes that were attached to a class. Reflection can also be used to find all the types in an assembly or dynamically locate and call functions in an assembly. It can even be used to emit the .NET intermediate language on the fly, to generate code that can be executed directly.

The documentation for the .NET Common Language Runtime contains more details on using reflection.

Listing All the Types in an Assembly

This 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 is run, 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 will list the members of a type.

```
using System;
using System.Reflection;
enum MyEnum
{
    Val1,
    Val2,
    Val3
}
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)
   {
        // Get the names and values in the enum
      Console.WriteLine("Fields of MyEnum");
      Type t = typeof (MyEnum);

        // create an instance of the enum
      object en = Activator.CreateInstance(t);
      foreach (FieldInfo f in t.GetFields(BindingFlags.LookupAll))
      {
         object o = f.GetValue(en);
         Console.WriteLine("{0}={1}", f, o);
      }

        // Now iterate through the fields of the class
      Console.WriteLine("Fields of MyClass");
      t = typeof (MyClass);
      foreach (MemberInfo m in t.GetFields(BindingFlags.LookupAll))
      {
         Console.WriteLine("{0}", m);
      }

        // and iterate through the methods of the class
      Console.WriteLine("Methods of MyClass");
      foreach (MethodInfo m in t.GetMethods(BindingFlags.LookupAll))
      {
         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 MyEnum
Int32 value__=0
MyEnum Val1=0
MyEnum Val2=1
MyEnum Val3=2
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 ()
```

To be able to obtain the value of a field in an enum, an instance of the enum must be present. While the enum could have been created using a simple `new` statement, the `Activator` class was used to illustrate how to create an instance on the fly.
When iterating over the methods in `MyClass`, the standard methods from `object` also show up.

Invoking Functions

In this example, reflection will be used to open the names of all the assemblies on the command lines, to search for the classes in them that implement a specific assembly, and then to create an instance of those classes and invoke a function on the assembly.

This is useful to provide a very late-bound architecture, where a component can be integrated with other components' runtime.
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 we'll search for.

```
// file=IProcess.cs
namespace MamaSoft
{
```

```
    interface IProcess
    {
        string Process(int param);
    }
}
```

Process1.cs

```
// 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

```
csc /target:library process1.cs iprocess.cs
```

Process2.cs

```
// 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

```
csc /target:library process2.cs iprocess.cs
```

Driver.cs

```
// 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

process process1.dll process2.dll

which will generate 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
```

Optimizations

The following optimizations are performed by the C# compiler when the `/optimize+` flag is used:

- Local variables that are never read are eliminated, even if they are 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

Chapter 32: Defensive Programming

Overview

THE .NET RUNTIME PROVIDES a few facilities to make programming less dangerous. Conditional methods and tracing can be used 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 only performs operations when compiled in a certain way. This is often used to add code that is only called when a debug build is made, and not called in other builds, usually because the additional check is too slow.
In C++, this would be done by using a macro in the `include` file that changed a function call to nothing if the debug symbol wasn't defined. This doesn't work in C#, however, because there is no `include` file or macro.
In C#, a method can be marked with the `Conditional` attribute, which indicates 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 out "`Bad State `" when it is run. If 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, and therefore can be used more liberally.
Calls to `Debug` are conditional on `DEBUG` being defined, and calls to `Trace` are conditional on `TRACE` being defined. By default, the VS IDE will define `TRACE` on both debug and retail builds, and `DEBUG` only on debug builds. When compiling from the command line, the appropriate option is required.
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 is false. The preceding code example would be written 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 bring up a dialog box, the code in `Main()` clears the `Listeners` collection and then adds a new listener that is hooked 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 asserts, the `Debug` and `Trace` classes can be used to send useful information to the current debug or trace listeners. This is a useful adjunct to running in the debugger, in that it is 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 could be changed at runtime to control the amount of logging that is performed.

```
// 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 last 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. What would be more useful is a way to set the value of such a variable externally.
The **`BooleanSwitch`** and **`TraceSwitch`** classes provide this feature. Their behavior can be controlled at runtime by either setting an environment variable or 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();
    }
}
```

In this example, an instance of **`BooleanSwitch`** is created as a static member of the class, and this variable is used to control whether output happens. If this code is run, it produces no output, but the **`debugOutput`** variable can be controlled by setting an environment variable.

```
set _Switch_MyClassDebugOutput=1
```

The environment variable name is created by prepending "_Switch_" in front of the display name (first parameter) of the constructor for **`BooleanSwitch`**. Running the code after setting this variable produces the following output:

```
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.

The value of a **`BooleanSwitch`** variable can also be set through the Windows Registry. For this example, a new **`DWORD`** value with the key

```
HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\COMPlus\Switches\MyClassDebugOutput
```

is created, and the **`DWORD`** value is set to **`0`** or **`1`** to set the value of the `BooleanSwitch`.

TraceSwitch

It is sometimes useful 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 four levels of information logging. They are defined in the **`TraceLevel`** enum.

LEVEL	NUMERIC VALUE
`Off`	0
`Error`	1
`Warning`	2
`Info`	3
`Verbose`	4

Each of the higher levels implies the lower level; if the level is set to **`Info`**, **`Error`** and **`Warning`** will also be set. The numeric values are used 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 would check 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(level);
        }
        set
        {
            level = value;
        }
    }
    public bool Mute
    {
        get
        {
            return(level == 0);
        }
    }
      public bool Terse
    {
        get
         {
            return((int) level >= (int) (SpecialSwitchLevel.Terse));
         }
    }
      public bool Verbose
    {
         get
        {
            return((int) level >= (int) SpecialSwitchLevel.Verbose);
        }
     }
       public bool Chatty
```

```
    {
        get
        {
          return((int) level >=(int) SpecialSwitchLevel.Chatty);
        }
    }

    protected override void SetSwitchSetting(int level)
    {
        if (level < 0)
            level = 0;
        if (level > 4)
            level = 4;

       this.level = (SpecialSwitchLevel) level;
    }

    SpecialSwitchLevel level;
}

class MyClass
{
      public MyClass(int i)
      {
            this.i = i;
      }
   [Conditional("DEBUG")]
   public void 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", "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();
   }
}
```

Chapter 33: The Command Line

Overview

THIS CHAPTER DESCRIBES the command-line switches that can be passed to the compiler. Options that can be abbreviated are shown with the abbreviated portion in brackets ([ ]).
The `/out` and `/target` options can be used more than once in a single compilation, and they apply only to those source files that follow the option.

Simple Usage

In the simple use, the following command-line command might be used:

csc test.cs
This will compile the file `test.cs` and produce a console assembly (.exe) that can then be executed. Multiple files may be specified 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 there are lots of files to compile, or complex options.

A response file is specified merely by listing it on the command line:

csc @<responsefile>

Multiple response files may be used on a single command line, or they may be mixed with options on the command line.

Command-Line Options

The following tables summarize the command-line options for the C# compiler. Most of these options can also be set from within the Visual Studio IDE.

Error Reporting Options

COMMAND	DESCRIPTION
`/warnaserror[+|-]`	Treat warnings as errors. When this option is on, the compiler will return an error code even if there were only warnings during the compilation

w[arn]:<level>	Set warning level (0-4)
/nowarn:<list>	Specify a comma-separated list of warnings to not report
/fullpaths	Specify the full path to a file in compilation errors or warnings

Input Options

COMMAND	DESCRIPTION
/addmodule:<file>	Specify modules that are part of this assembly
/codepage:<id>	Use the specified code page id to open source files
/nostdlib[+|-]	Do not import the standard library (mscorlib.dll). This might be used to switch to a different standard library for a specific target device
/recurse:<filespec>	Search subdirectories for files to compile
/r[eference]:<file>	Specify metadata file to import

Output Options

COMMAND	DESCRIPTION
/a[ssembly] [+|-]	Emit an assembled PE
/o[ptimize] [+|-]	Enable optimizations
/out:<outfile>	Set output filename
/t[arget]:module	Create module that can be added to another assembly

/t[arget]:library	Create a library instead of an application
/t[arget]:exe	Create a console application (default)
/t[arget]:winexe	Create a Windows GUI application
/nooutput[+|-]	Only check code for errors; do not emit executable
/baseaddress:<addr>	Specify the library base address

Processing Options

COMMAND	DESCRIPTION
/debug[+|-]	Emit debugging information
/incr[emental] [+|-]	Perform an incremental build
/checked[+|-]	Check for overflow and under-flow by default
/unsafe[+|-]	Allow “unsafe” code
/d[efine]:<def-list>	Define conditional compilation symbol(s)
/doc:<file>	Specify a file to store XML Doc-Comments into
/win32res:<resfile>	Specify a Win32 resource file
/win32icon:<iconfile>	Specify a Win32 icon file
/res[ource]:<file>[,<name>[,<MIMEtype]]	Embeds a resource into this assembly
/linkres[ource] :<file>[,<name>[,<MIMEtype]]	Link a resource into this assembly without embedding it

Miscellaneous

COMMAND	DESCRIPTION
`/? or /help`	Display the usage message
`/nologo`	Do not display the compiler copyright banner
`/bugreport:<file>`	Create report file
`/main:<classname>`	Specify the class to use for the `Main()` entry point

Chapter 34: C# Compared to Other Languages

Overview

THIS CHAPTER WILL COMPARE 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.

There is also a section of this chapter that discusses the .NET versions of Visual C++ and Visual Basic, since they are also somewhat different than their predecessors.

Differences Between C# and C/C++

C# code will be familiar to C and C++ programmers, but there are a few big differences and a number of small differences. The following gives an overview of the differences. For a more detailed perspective, see the Microsoft white paper, "C# for the C++ Programmer."

A Managed Environment

C# runs in the .NET Runtime environment. This not only means that there are many things that aren't under the programmer's control, it also provides a brand- new set of frameworks. Together, this means a few things are changed.

- Object deletion is performed by the garbage collector sometime after the object is no longer used. Destructors (a.k.a. finalizers) can be used for some cleanup, but not in the way that C++ destructors are used.
- There are no pointers in the C# language. Well, there are in `unsafe` mode, but they are rarely used. References are used instead, and they are 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 intermediate language, IL) and metadata to describe that compiled code. All .NET languages query the metadata to determine the same information that is contained in C++ .h files, and the include files are therefore absent.
- Calling native code requires a bit more work.
- There is no C/C++ Runtime library. The same things—such as string manipulation, file I/O, and other routines—can be done with the .NET Runtime and are found in the namespaces that start with `System`.
- Exception handling is used instead of error returns.

.NET Objects

C# objects all have the ultimate base class `object`, and there is only single inheritance of classes, though there is multiple implementation of interfaces.

Lightweight objects, such as data types, can be declared as structs (also known as value types), which means they are allocated on the stack instead of the heap.

C# structs and other value types (including the built-in data types) can be used in situations where objects are required by boxing them, which automatically copies the value into a heap-allocated wrapper that is 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# provides the `params` keyword to provide support similar to varags.

C# Statements

C# statements have high fidelity to C++ statements. There are a few notable differences:

- The new keyword means "obtain a new copy of." The object is heap-allocated if it is a reference type, and stack or inline allocated if it is a value type.
- All statements that test a Boolean condition now require a variable of type `bool`. There is no automatic conversion from `int` to `bool`, so "`if (i)`" isn't valid.
- Switch statements disallow fall-through, to reduce errors. Switch can also be used on string values.
- `Foreach` can be used to iterate over objects and collections.
- `Checked` and `unchecked` are used to control whether arithmetic operations and conversions are checked for overflow.
- Definite assignment requires that objects have a definite value before being used.

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. Attribute information is retrieved through a process known as reflection.

Attributes are written inside of square brackets, and can be placed on classes, members, parameters, and other code elements. Here's an example:

```
[CodeReview("1/1/199", 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 while there is preprocessor support for conditional code, there is 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, there is no order dependence in C# code, and no forward declarations. The order of classes in source files is unimportant; classes can be rearranged 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

Differences Between C# and Java

C# and Java have similar roots, so it's no surprise that there are similarities between them. There are a fair number of differences between them, however. The biggest difference is that C# sits on the .NET Frameworks and Runtime, and Java sits on the Java Frameworks and Runtime.

Data Types

C# has more primitive data types than Java. The following table summarizes the Java types and their C# analogs:

C# TYPE	JAVA TYPE	COMMENT
sbyte	byte	C# byte is unsigned
short	short	
int	int	
long	long	
bool	Boolean	
float	float	
double	double	
char	char	
string	string	
object	object	
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 put into an instance of a wrapper class, and the wrap-per class put in that collection.
C# approaches this problem differently. In C#, primitive types are stack-allocated as in Java, but they are also considered to derived from the ultimate base class, `object`. This means that the primitive types can have member functions defined and called on them. In other words, the following code can be written:

```
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 were an object. This works well when the compiler knows it's dealing with a primitive, but 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. Declaring the object variable is done for illustration only; in real code, `v` would be passed directly, and the boxing would happen at the call site. The boxed integer can be extracted 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 are 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 the built-in types can be written as constants.

To make user-defined types behave naturally, C# structs can overload arithmetic operators so that numeric operations can be performed, and conversions so that implicit and explicit conversions can be performed between structs and other types. C# also supports overloading on classes as well.

A `struct` is written using the same syntax as a `class`, except that a struct cannot 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 only be set 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 Control(int value)
   {
      this.value = value;
   }
      public Control() : base(value)
   {
   }
   int value;
}
```

Static Constructors

Instead of 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 non-virtual by default, and `virtual` must be specified explicitly to make a function virtual. Because of this, there are no final methods in C#, though the equivalent of a final class can be achieved using `sealed`.

C# provides better versioning support than Java, and this results in a few small changes. Method overloading is done by name rather than by signature, which means that the addition of classes in a base class will not change program 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 is 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 the versioning semantics be specified explicitly. If `Process()` had been a virtual function in the derived class, Java would assume that any base class function that matched in sig- nature would be a base for that virtual, which is unlikely to be correct.
In C#, virtual functions are only overridden if the `override` keyword is specified. See Chapter 11, "Versioning Using New and Override," for more information.

Accessibility of Members

In addition to `public`, `private`, and `protected` accessibility, C# adds `internal`. Members with `internal` accessibility can be accessed from other classes within the same project, but not from outside the project.

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 exactly the same as `ref`, except that the variable passed as the parameter doesn't have to have a known value before the call.

Identifying Types

Java uses the `GetClass()` method to return a `Class` object, which contains information about the object on which it is called. The `Type` object is the .NET analog to the `Class` object and can be obtained 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

While 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 it can also be used to hide interface implementations from the user. For more information, see Chapter 10, "Interfaces."

Properties and Indexers

The property idiom is often used in Java programs 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 is used to specify how the object must be formed, and a method in that interface is called for the callback. A similar approach can be used in C# with interfaces.

C# adds delegates, which can be thought of as typesafe 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 Frameworks. 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 is fired.

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. Attribute information is retrieved through a process known as reflection.

Attributes are written inside of square brackets, and can be placed on classes, members, parameters, and other code elements. Here's an example:

```
[CodeReview("1/1/199", 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 is used to locate a package and import the types into the current file. In C#, this operation is split. The assemblies that a section of code relies upon must be explicitly specified, 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 they must be specified using their fully qualified name. For example, the regular expression class is named `System.Text.RegularExpressions.Regex`. That class name could be used directly, or a `using` statement could be used to import the types in a namespace to the top-level namespace. With the following using clause

```
using System.Text.RegularExpressions;
```

the class can be specified merely by using `Regex`. There is also a variant of the `using` statement that allows aliases for types to be specified if there is a name collision.

Overflows

Java doesn't detect overflow on conversions or mathematical errors.
In C#, the detection of these can be controlled 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 default context is controlled by the `/checked` compiler flag.

Unsafe Code

Unsafe code in C# allows the use of pointer variables, and it is used 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 cannot be verified to be safe by the runtime, it can only be executed if it is fully trusted by the runtime. This prevents execution in download scenarios.

Strings

The C# string object can be indexed to access specific characters. Comparison between strings performs a comparison of 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.

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

There are a few miscellaneous differences:

- 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 is no labeled break statement; `goto` replaces it.
- The `switch` statement prohibits fall-through, and `switch` can be used on string variables.
- There is 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 VB6 has only limited object-oriented features. VB7 adds additional object- oriented features to the VB language, and it may be instructive to also study the VB7 documentation.

Code Appearance

In VB, statement blocks are ended with some sort of `END` statement, and there can't be multiple statements on a single line. In C#, blocks are denoted using braces {}, and the location of line breaks doesn't matter, as the end of a statement is indicated by a semicolon. Though it might be bad form and ugly to read, in C# the following can be written:

```
for (int j = 0; j < 10; j++) {if (j == 5) Func(j); else return;}
```

That line will mean 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

While there is a considerable amount of overlap in data types between VB and C#, there are some important differences, and a similar name may mean a different data type.
The most important difference is that C# is more strict on variable declaration and usage. All variables must be declared before they are used, and they must be declared with a specific type—there is no `Variant` type that can hold any type. [1]
Variable declarations are made simply by using the name of the type before the variable; there is no `dim` statement.

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, the `checked` statement can be used to turn on the detection of overflow. See Chapter 15, "Conversions," for more information.

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 is a `long` type as well, but it is 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 bigger and much slower. `Byte`, how- ever, 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 they can't be used by all other languages in .NET, so they should only be used 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 do not exist for the C# string type. There are functions to search strings, extract substrings, and perform other operations; see the documentation for the `System.String` type for details.
String concatenation is performed using the `+` operator rather than the `&` operator.

Arrays

In C#, the first element of an array is always index 0, and there is no way to set upper or lower bounds, and no way to `redim` an array. There is, however, an `ArrayList` in the `System.Collection` namespace that does allow resizing, along with 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.

VB OPERATOR	C# EQUIVALENT
^	None. See Math.Pow()

Mod	%
&	+
=	==
<>	!=
Like	None. System.Text.RegularExpressions.Regex does some of this, but it is more complex
Is	None. The C# `is` operator means something different
And	&&
Or	||
Xor	^
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 are always associated with a specific class. This results in code that is structured and organized quite differently than VB code, but there are still some common elements.

Properties can still be used, though they have a different syntax and there are no default properties.

Functions

In C#, function parameters must have a declared type, and `ref` is used instead of `ByVal` to indicate that the value of a passed variable may be modified. The `ParamArray` function can be achieved 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

In C#, there is 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 there is an optional `else` statement.

The following VB code

```
If size < 60 Then
   value = 50
Else
   value = 55
   order = 12
End If
```

can be rewritten as

```
if (size < 60)
   value = 50;
```

```
else
{
    value = 55;
    order = 12;
}
```

There is no `ElseIf` statement in C#.

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 the following VB code

```
For i = 1 To 100
    ' other code here
}
```

can be rewritten as

```
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 is used to loop 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. The following VB code

```
I = 1
fact = 1
Do While I <= n
    fact = fact * I
    I = I + 1
Loop
```

can be rewritten as:

```
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`. This VB code

```
Select Case x
    Case 1
```

```
        Func1
    Case 2
        Func2
    Case 3
        Func2
    Case Else
        Func3
End Select
```

can be rewritten as:

```
switch (x)
{
    case 1:
        Func1();
        break;
    case 2:
    case 3:
        Func2();
        break;
    default:
        Func3();
        break;
}
```

On Error

There is no `On Error` statement in C#. Error conditions in .NET are communicated through exceptions. See Chapter 4, "Exception Handling," for more details.

Missing Statements

There is no `With`, `Choose`, or the equivalent of `Switch` in C#. There is also no `CallByName` feature, though this can be performed through reflection.

[1]The `object` type can contain any type, but it knows exactly what type it contains.

[2]See Chapter 1, “Object-Oriented Basics,” for more information.

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 Common Language Runtime. The Visual C++ model allows the programmer more control than the C# model, in that the user is allowed to write both managed (garbage- collected) and unmanaged (using `new` and `delete`) objects.
A .NET component is created by using keywords to modify the meaning of existing C++ constructs. For example, when the `__gc` keyword is placed in front of a class definition, it enables the creation of a managed class and restricts the class from using constructs that cannot be expressed in the .NET world (such as multiple inheritance). The .NET system classes can also be used from the managed extensions.

Visual Basic has also seen considerable improvements. It now has object-oriented concepts such as inheritance, encapsulation, and overloading, which allow it to operate well in the .NET world.

Chapter 35: C# Futures

AS MENTIONED AT THE BEGINNING of the book, C# is an evolving language, and it is therefore difficult to speculate on the future of the language except where Microsoft has an official position.

One feature that Microsoft is working on is generics, which are generic versions of templates. If generics were present in the language, it would be possible to write strongly typed collection classes, such as a stack that could hold only a specific type, rather than any object.
If such a stack class existed, it could be used with the `int` type, and the stack could only contain `int` values. This has two big benefits:

- When the programmer tries to pull a `float` off a stack that stores `int`, the current collections will report this error at runtime. Generics would allow it to be reported at compile time.
- In current collections, all value types must be boxed, and `int` values are therefore stored as reference objects rather than value objects. As a result, adding and removing objects imposes overhead, which would be absent if there was generic support.

List of Figures

List of Tables

List of Sidebars