Special Annotated Edition for C# 4.0

The C# Programming Language

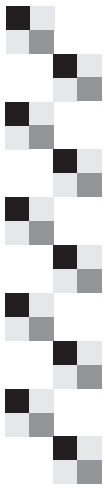
Fourth Edition

Anders Hejlsberg
Mads Torgersen
Scott Wiltamuth
Peter Golde

www.it-ebooks.info

The C# Programming Language
Fourth Edition

www.it-ebooks.info



The C# Programming Language

Fourth Edition

Anders Hejlsberg
Mads Torgersen
Scott Wiltamuth
Peter Golde

Addison-Wesley

Upper Saddle River, NJ • Boston • Indianapolis • San Francisco
New York • Toronto • Montreal • London • Munich • Paris • Madrid
Capetown • Sydney • Tokyo • Singapore • Mexico City

www.it-ebooks.info

Many of the designations used by manufacturers and sellers to distinguish their products are claimed as trademarks. Where those designations appear in this book, and the publisher was aware of a trademark claim, the designations have been printed with initial capital letters or in all capitals.

The .NET logo is either a registered trademark or trademark of Microsoft Corporation in the United States and/or other countries and is used under license from Microsoft.

Microsoft, Windows, Visual Basic, Visual C#, and Visual C++ are either registered trademarks or trademarks of Microsoft Corporation in the U.S.A. and/or other countries/regions.

The authors and publisher have taken care in the preparation of this book, but make no expressed or implied warranty of any kind and assume no responsibility for errors or omissions. No liability is assumed for incidental or consequential damages in connection with or arising out of the use of the information or programs contained herein.

The publisher offers excellent discounts on this book when ordered in quantity for bulk purchases or special sales, which may include electronic versions and/or custom covers and content particular to your business, training goals, marketing focus, and branding interests. For more information, please contact:

U.S. Corporate and Government Sales
(800) 382-3419
corpsales@pearsontechgroup.com

For sales outside the United States, please contact:

International Sales
international@pearson.com

Visit us on the Web: informit.com/aw

Library of Congress Cataloging-in-Publication Data

The C# programming language / Anders Hejlsberg ... [et al.]. — 4th ed.
p. cm.
Includes index.
ISBN 978-0-321-74176-9 (hardcover : alk. paper)
1. C# (Computer program language) I. Hejlsberg, Anders.
QA76.73.C154H45 2010
005.13'3—dc22

2010032289

Copyright © 2011 Microsoft Corporation

All rights reserved. Printed in the United States of America. This publication is protected by copyright, and permission must be obtained from the publisher prior to any prohibited reproduction, storage in a retrieval system, or transmission in any form or by any means, electronic, mechanical, photocopying, recording, or likewise. For information regarding permissions, write to:

Pearson Education, Inc.
Rights and Contracts Department
501 Boylston Street, Suite 900
Boston, MA 02116
Fax: (617) 671-3447

ISBN-13: 978-0-321-74176-9
ISBN-10: 0-321-74176-5
Text printed in the United States on recycled paper at Courier in Westford, Massachusetts.
First printing, October 2010

www.it-ebooks.info

Contents

Foreword *xi*
Preface *xiii*
About the Authors *xv*
About the Annotators *xvii*

1 Introduction 1
1.1 Hello, World 3
1.2 Program Structure 4
1.3 Types and Variables 6
1.4 Expressions 13
1.5 Statements 16
1.6 Classes and Objects 21
1.7 Structs 50
1.8 Arrays 53
1.9 Interfaces 56
1.10 Enums 58
1.11 Delegates 60
1.12 Attributes 61

2 Lexical Structure 65
2.1 Programs 65
2.2 Grammars 65
2.3 Lexical Analysis 67
2.4 Tokens 71
2.5 Preprocessing Directives 85

www.it-ebooks.info

3 **Basic Concepts 99**
- 3.1 Application Start-up 99
- 3.2 Application Termination 100
- 3.3 Declarations 101
- 3.4 Members 105
- 3.5 Member Access 107
- 3.6 Signatures and Overloading 117
- 3.7 Scopes 120
- 3.8 Namespace and Type Names 127
- 3.9 Automatic Memory Management 132
- 3.10 Execution Order 137

4 **Types 139**
- 4.1 Value Types 140
- 4.2 Reference Types 152
- 4.3 Boxing and Unboxing 155
- 4.4 Constructed Types 160
- 4.5 Type Parameters 164
- 4.6 Expression Tree Types 165
- 4.7 The dynamic Type 166

5 **Variables 169**
- 5.1 Variable Categories 169
- 5.2 Default Values 175
- 5.3 Definite Assignment 176
- 5.4 Variable References 192
- 5.5 Atomicity of Variable References 193

6 **Conversions 195**
- 6.1 Implicit Conversions 196
- 6.2 Explicit Conversions 204
- 6.3 Standard Conversions 213
- 6.4 User-Defined Conversions 214
- 6.5 Anonymous Function Conversions 219
- 6.6 Method Group Conversions 226

www.it-ebooks.info

7 Expressions 231
7.1 Expression Classifications 231
7.2 Static and Dynamic Binding 234
7.3 Operators 238
7.4 Member Lookup 247
7.5 Function Members 250
7.6 Primary Expressions 278
7.7 Unary Operators 326
7.8 Arithmetic Operators 331
7.9 Shift Operators 343
7.10 Relational and Type-Testing Operators 344
7.11 Logical Operators 355
7.12 Conditional Logical Operators 358
7.13 The Null Coalescing Operator 360
7.14 Conditional Operator 361
7.15 Anonymous Function Expressions 364
7.16 Query Expressions 373
7.17 Assignment Operators 389
7.18 Expression 395
7.19 Constant Expressions 395
7.20 Boolean Expressions 397

8 Statements 399
8.1 End Points and Reachability 400
8.2 Blocks 402
8.3 The Empty Statement 404
8.4 Labeled Statements 406
8.5 Declaration Statements 407
8.6 Expression Statements 412
8.7 Selection Statements 413
8.8 Iteration Statements 420
8.9 Jump Statements 429
8.10 The try Statement 438
8.11 The checked and unchecked Statements 443
8.12 The lock Statement 443
8.13 The using Statement 445
8.14 The yield Statement 449

www.it-ebooks.info

9 Namespaces 453
9.1 Compilation Units 453
9.2 Namespace Declarations 454
9.3 Extern Aliases 456
9.4 Using Directives 457
9.5 Namespace Members 463
9.6 Type Declarations 464
9.7 Namespace Alias Qualifiers 464

10 Classes 467
10.1 Class Declarations 467
10.2 Partial Types 481
10.3 Class Members 490
10.4 Constants 506
10.5 Fields 509
10.6 Methods 520
10.7 Properties 545
10.8 Events 559
10.9 Indexers 566
10.10 Operators 571
10.11 Instance Constructors 579
10.12 Static Constructors 586
10.13 Destructors 589
10.14 Iterators 592

11 Structs 607
11.1 Struct Declarations 608
11.2 Struct Members 609
11.3 Class and Struct Differences 610
11.4 Struct Examples 619

12 Arrays 625
12.1 Array Types 625
12.2 Array Creation 628
12.3 Array Element Access 628

www.it-ebooks.info

12.4 Array Members 628
12.5 Array Covariance 629
12.6 Array Initializers 630

13 Interfaces 633
13.1 Interface Declarations 633
13.2 Interface Members 639
13.3 Fully Qualified Interface Member Names 645
13.4 Interface Implementations 645

14 Enums 663
14.1 Enum Declarations 663
14.2 Enum Modifiers 664
14.3 Enum Members 665
14.4 The System.Enum Type 668
14.5 Enum Values and Operations 668

15 Delegates 671
15.1 Delegate Declarations 672
15.2 Delegate Compatibility 676
15.3 Delegate Instantiation 676
15.4 Delegate Invocation 677

16 Exceptions 681
16.1 Causes of Exceptions 683
16.2 The System.Exception Class 683
16.3 How Exceptions Are Handled 684
16.4 Common Exception Classes 685

17 Attributes 687
17.1 Attribute Classes 688
17.2 Attribute Specification 692
17.3 Attribute Instances 698
17.4 Reserved Attributes 699
17.5 Attributes for Interoperation 707

www.it-ebooks.info

18 **Unsafe Code** 709
- 18.1 Unsafe Contexts 710
- 18.2 Pointer Types 713
- 18.3 Fixed and Moveable Variables 716
- 18.4 Pointer Conversions 717
- 18.5 Pointers in Expressions 720
- 18.6 The fixed Statement 728
- 18.7 Fixed-Size Buffers 733
- 18.8 Stack Allocation 736
- 18.9 Dynamic Memory Allocation 738

A **Documentation Comments** 741
- A.1 Introduction 741
- A.2 Recommended Tags 743
- A.3 Processing the Documentation File 754
- A.4 An Example 760

B **Grammar** 767
- B.1 Lexical Grammar 767
- B.2 Syntactic Grammar 777
- B.3 Grammar Extensions for Unsafe Code 809

C **References** 813

Index 815

www.it-ebooks.info

Foreword

It's been ten years since the launch of .NET in the summer of 2000. For me, the significance of .NET was the one-two combination of managed code for local execution and XML messaging for program-to-program communication. What wasn't obvious to me at the time was how important C# would become.

From the inception of .NET, C# has provided the primary lens used by developers for understanding and interacting with .NET. Ask the average .NET developer the difference between a value type and a reference type, and he or she will quickly say, "Struct versus class," not "Types that derive from `System.ValueType` versus those that don't." Why? Because people use languages—not APIs—to communicate their ideas and intention to the runtime and, more importantly, to each other.

It's hard to overstate how important having a great language has been to the success of the platform at large. C# was initially important to establish the baseline for how people think about .NET. It's been even more important as .NET has evolved, as features such as iterators and true closures (also known as anonymous methods) were introduced to developers as purely language features implemented by the C# compiler, not as features native to the platform. The fact that C# is a vital center of innovation for .NET became even more apparent with C# 3.0, with the introduction of standardized query operators, compact lambda expressions, extension methods, and runtime access to expression trees—again, all driven by development of the language and compiler. The most significant feature in C# 4.0, dynamic invocation, is also largely a feature of the language and compiler rather than changes to the CLR itself.

It's difficult to talk about C# without also talking about its inventor and constant shepherd, Anders Hejlsberg. I had the distinct pleasure of participating in the recurring C# design meetings for a few months during the C# 3.0 design cycle, and it was enlightening watching Anders at work. His instinct for knowing what developers will and will not like is truly

www.it-ebooks.info

world-class—yet at the same time, Anders is extremely inclusive of his design team and manages to get the best design possible.

With C# 3.0 in particular, Anders had an uncanny ability to take key ideas from the functional language community and make them accessible to a very broad audience. This is no trivial feat. Guy Steele once said of Java, "We were not out to win over the Lisp programmers; we were after the C++ programmers. We managed to drag a lot of them about halfway to Lisp." When I look at C# 3.0, I think C# has managed to drag at least one C++ developer (me) most of the rest of the way. C# 4.0 takes the next step toward Lisp (and JavaScript, Python, Ruby, et al.) by adding the ability to cleanly write programs that don't rely on static type definitions.

As good as C# is, people still need a document written in both natural language (English, in this case) and some formalism (BNF) to grok the subtleties and to ensure that we're all speaking the same C#. The book you hold in your hands is that document. Based on my own experience, I can safely say that every .NET developer who reads it will have at least one "aha" moment and will be a better developer for it.

Enjoy.

Don Box
Redmond, Washington
May 2010

www.it-ebooks.info

Preface

The C# project started more than 12 years ago, in December 1998, with the goal to create a simple, modern, object-oriented, and type-safe programming language for the new and yet-to-be-named .NET platform. Since then, C# has come a long way. The language is now in use by more than a million programmers and has been released in four versions, each with several major new features added.

This book, too, is in its fourth edition. It provides a complete technical specification of the C# programming language. This latest edition includes two kinds of new material not found in previous versions. Most notably, of course, it has been updated to cover the new features of C# 4.0, including dynamic binding, named and optional parameters, and covariant and contravariant generic types. The overarching theme for this revision has been to open up C# more to interaction with objects outside of the .NET environment. Just as LINQ in C# 3.0 gave a language-integrated feel to code used to access external data sources, so the dynamic binding of C# 4.0 makes the interaction with objects from, for example, dynamic programming languages such as Python, Ruby, and JavaScript feel native to C#.

The previous edition of this book introduced the notion of annotations by well-known C# experts. We have received consistently enthusiastic feedback about this feature, and we are extremely pleased to be able to offer a new round of deep and entertaining insights, guidelines, background, and perspective from both old and new annotators throughout the book. We are very happy to see the annotations continue to complement the core material and help the C# features spring to life.

Many people have been involved in the creation of the C# language. The language design team for C# 1.0 consisted of Anders Hejlsberg, Scott Wiltamuth, Peter Golde, Peter Sollich, and Eric Gunnerson. For C# 2.0, the language design team consisted of Anders Hejlsberg, Peter Golde, Peter Hallam, Shon Katzenberger, Todd Proebsting, and Anson Horton.

www.it-ebooks.info

Furthermore, the design and implementation of generics in C# and the .NET Common Language Runtime is based on the "Gyro" prototype built by Don Syme and Andrew Kennedy of Microsoft Research. C# 3.0 was designed by Anders Hejlsberg, Erik Meijer, Matt Warren, Mads Torgersen, Peter Hallam, and Dinesh Kulkarni. On the design team for C# 4.0 were Anders Hejlsberg, Matt Warren, Mads Torgersen, Eric Lippert, Jim Hugunin, Lucian Wischik, and Neal Gafter.

It is impossible to acknowledge the many people who have influenced the design of C#, but we are nonetheless grateful to all of them. Nothing good gets designed in a vacuum, and the constant feedback we receive from our large and enthusiastic community of developers is invaluable.

C# has been and continues to be one of the most challenging and exciting projects on which we've worked. We hope you enjoy using C# as much as we enjoy creating it.

Anders Hejlsberg
Mads Torgersen
Scott Wiltamuth
Peter Golde
Seattle, Washington
September 2010

www.it-ebooks.info

About the Authors

Anders Hejlsberg is a programming legend. He is the architect of the C# language and a Microsoft Technical Fellow. He joined Microsoft Corporation in 1996, following a 13-year career at Borland, where he was the chief architect of Delphi and Turbo Pascal.

Mads Torgersen is the program manager for the C# language at Microsoft Corporation, where he runs the day-to-day language design process and maintains the language specification.

Scott Wiltamuth is director of program management for the Visual Studio Professional team at Microsoft Corporation. At Microsoft, he has worked on a wide range of development tools, including OLE Automation, Visual Basic, Visual Basic for Applications, VBScript, JScript, Visual J++, and Visual C#.

Peter Golde was the lead developer of the original Microsoft C# compiler. As the primary Microsoft representative on the ECMA committee that standardized C#, he led the implementation of the compiler and worked on the language design. He is currently an architect at Microsoft Corporation working on compilers.

www.it-ebooks.info

This page intentionally left blank

www.it-ebooks.info

About the Annotators

Brad Abrams was a founding member of both the Common Language Runtime and the .NET Framework teams at Microsoft Corporation, where he was most recently the director of program management for WCF and WF. Brad has been designing parts of the .NET Framework since 1998, when he started his framework design career building the BCL (Base Class Library) that ships as a core part of the .NET Framework. Brad graduated from North Carolina State University in 1997 with a BS in computer science. Brad's publications include: *Framework Design Guidelines, Second Edition* (Addison-Wesley, 2009), and *.NET Framework Standard Library Annotated Reference* (Volumes 1 and 2) (Addison-Wesley, 2006).

Joseph Albahari is coauthor of *C# 4.0 in a Nutshell* (O'Reilly, 2007), the *C# 3.0 Pocket Reference* (O'Reilly, 2008), and the *LINQ Pocket Reference* (O'Reilly, 2008). He has 17 years of experience as a senior developer and software architect in the health, education, and telecommunication industries, and is the author of LINQPad, the utility for interactively querying databases in LINQ.

Krzysztof Cwalina is a principal architect on the .NET Framework team at Microsoft. He started his career at Microsoft designing APIs for the first release of the Framework. Currently, he is leading the effort to develop, promote, and apply design and architectural standards to the development of the .NET Framework. He is a coauthor of *Framework Design Guidelines* (Addison-Wesley, 2005). Reach him at his blog at http://blogs.msdn.com/kcwalina.

Jesse Liberty ("Silverlight Geek") is a senior program manager at Microsoft and the author of numerous best-selling programming books and dozens of popular articles. He's also a frequent speaker at events world-wide. His blog,http://JesseLiberty.com, is required reading for Silverlight, WPF, and Windows Phone 7 developers. Jesse has more than two decades of real-world programming experience, including stints as a vice president at Citi and as a distinguished software engineer at AT&T. He can be reached through his blog and followed at @JesseLiberty.

www.it-ebooks.info

Eric Lippert is a senior developer on the C# compiler team at Microsoft. He has worked on the design and implementation of the Visual Basic, VBScript, JScript, and C# languages and Visual Studio Tools For Office. His blog about all those topics and more can be found at http://blogs.msdn.com/EricLippert.

Christian Nagel is a Microsoft regional director and MVP. He is the author of several books, including *Professional C# 4 with .NET 4* (Wrox, 2010) and *Enterprise Services with the .NET Framework* (Addison-Wesley, 2005). As founder of CN innovation and associate of thinktecture, he teaches and coaches software developers on various Microsoft .NET technologies. Christian can be reached at http://www.cninnovation.com.

Vladimir Reshetnikov is a Microsoft MVP for Visual C#. He has more than eight years of software development experience, and about six years of experience in Microsoft .NET and C#. He can be reached at his blog http://nikov-thoughts.blogspot.com.

Marek Safar is the lead developer of the Novell C# compiler team. He has been working on most of the features of Mono C# compiler over the past five years. Reach him at his blog at http://mareksafar.blogspot.com.

Chris Sells is a program manager for the Business Platform Division (aka the SQL Server division) of Microsoft Corporation. He's written several books, including *Programming WPF* (O'Reilly, 2007), *Windows Forms 2.0 Programming* (Addison-Wesley, 2006), and *ATL Internals* (Addison-Wesley, 1999). In his free time, Chris hosts various conferences and makes a pest of himself on Microsoft internal product team discussion lists. More information about Chris, and his various projects, is available at http://www.sellsbrothers.com.

Peter Sestoft is a professor of software development at the IT University of Copenhagen, Denmark. He was a member of the ECMA International C# standardization committee from 2003 through 2006, and is the author of *C# Precisely* (MIT Press, 2004) and *Java Precisely* (MIT Press, 2005). Find him at http://www.itu.dk/people/sestoft.

Jon Skeet is the author of *C# in Depth* (Manning, 2010) and a C# MVP. He works for Google in London, writing and speaking about C# in his leisure time. His blog is at http://msmvps.com/jon.skeet —or you can find him answering questions most days on Stack Overflow (http://stackoverflow.com).

Bill Wagner is the founder of SRT Solutions, a Microsoft regional director, and a C# MVP. He spent the overwhelming majority of his professional career between curly braces. He is the author of *Effective C#* (Addison-Wesley, 2005) and *More Effective C#* (Addison-Wesley, 2009), a former C# columnist for *Visual Studio Magazine*, and a contributor to the C# Developer Center on MSDN. You can keep up with his evolving thoughts on C# and other topics at http://srtsolutions.com/blogs/billwagner.

www.it-ebooks.info

1. Introduction

C# (pronounced "See Sharp") is a simple, modern, object-oriented, and type-safe programming language. C# has its roots in the C family of languages and will be immediately familiar to C, C++, and Java programmers. C# is standardized by ECMA International as the ***ECMA-334*** standard and by ISO/IEC as the ***ISO/IEC 23270*** standard. Microsoft's C# compiler for the .NET Framework is a conforming implementation of both of these standards.

C# is an object-oriented language, but C# further includes support for ***component-oriented*** programming. Contemporary software design increasingly relies on software components in the form of self-contained and self-describing packages of functionality. Key to such components is that they present a programming model with properties, methods, and events; they have attributes that provide declarative information about the component; and they incorporate their own documentation. C# provides language constructs to directly support these concepts, making C# a very natural language in which to create and use software components.

Several C# features aid in the construction of robust and durable applications: ***Garbage collection*** automatically reclaims memory occupied by unused objects; ***exception handling*** provides a structured and extensible approach to error detection and recovery; and the ***type-safe*** design of the language makes it impossible to read from uninitialized variables, to index arrays beyond their bounds, or to perform unchecked type casts.

C# has a ***unified type system***. All C# types, including primitive types such as `int` and `double`, inherit from a single root `object` type. Thus all types share a set of common operations, and values of any type can be stored, transported, and operated upon in a consistent manner. Furthermore, C# supports both user-defined reference types and value types, allowing dynamic allocation of objects as well as in-line storage of lightweight structures.

To ensure that C# programs and libraries can evolve over time in a compatible manner, much emphasis has been placed on ***versioning*** in C#'s design. Many programming languages pay little attention to this issue. As a result, programs written in those languages break more often than necessary when newer versions of dependent libraries are introduced. Aspects of C#'s design that were directly influenced by versioning considerations include the separate `virtual` and `override` modifiers, the rules for method overload resolution, and support for explicit interface member declarations.

www.it-ebooks.info

The rest of this chapter describes the essential features of the C# language. Although later chapters describe rules and exceptions in a detail-oriented and sometimes mathematical manner, this chapter strives for clarity and brevity at the expense of completeness. The intent is to provide the reader with an introduction to the language that will facilitate the writing of early programs and the reading of later chapters.

CHRIS SELLS I'm absolutely willing to go with "modern, object-oriented, and type-safe," but C# isn't nearly as simple as it once was. However, given that the language gained functionality such as generics and anonymous delegates in C# 2.0, LINQ-related features in C# 3.0, and dynamic values in C# 4.0, the programs themselves become simpler, more readable, and easier to maintain—which should be the goal of any programming language.

ERIC LIPPERT C# is also increasingly a functional programming language. Features such as type inference, lambda expressions, and monadic query comprehensions allow traditional object-oriented developers to use these ideas from functional languages to increase the expressiveness of the language.

CHRISTIAN NAGEL C# is not a pure object-oriented language but rather a language that is extended over time to get more productivity in the main areas where C# is used. Programs written with C# 3.0 can look completely different than programs written in C# 1.0 with functional programming constructs.

JON SKEET Certain aspects of C# have certainly made this language more functional over time—but at the same time, mutability was encouraged in C# 3.0 by both automatically implemented properties and object initializers. It will be interesting to see whether features encouraging immutability arrive in future versions, along with support for other areas such as tuples, pattern matching, and tail recursion.

BILL WAGNER This section has not changed since the first version of the C# spec. Obviously, the language has grown and added new idioms—and yet C# is still an approachable language. These advanced features are always within reach, but not always required for every program. C# is still approachable for inexperienced developers even as it grows more and more powerful.

www.it-ebooks.info

1.1 Hello, World

The "Hello, World" program is traditionally used to introduce a programming language. Here it is in C#:

```
using System;

class Hello
{
    static void Main() {
        Console.WriteLine("Hello, World");
    }
}
```

C# source files typically have the file extension `.cs`. Assuming that the "Hello, World" program is stored in the file `hello.cs`, the program can be compiled with the Microsoft C# compiler using the command line

```
csc hello.cs
```

which produces an executable assembly named `hello.exe`. The output produced by this application when it is run is

```
Hello, World
```

The "Hello, World" program starts with a `using` directive that references the `System` namespace. Namespaces provide a hierarchical means of organizing C# programs and libraries. Namespaces contain types and other namespaces—for example, the `System` namespace contains a number of types, such as the `Console` class referenced in the program, and a number of other namespaces, such as `IO` and `Collections`. A `using` directive that references a given namespace enables unqualified use of the types that are members of that namespace. Because of the `using` directive, the program can use `Console.WriteLine` as shorthand for `System.Console.WriteLine`.

The `Hello` class declared by the "Hello, World" program has a single member, the method named `Main`. The `Main` method is declared with the `static` modifier. While instance methods can reference a particular enclosing object instance using the keyword `this`, static methods operate without reference to a particular object. By convention, a static method named `Main` serves as the entry point of a program.

The output of the program is produced by the `WriteLine` method of the `Console` class in the `System` namespace. This class is provided by the .NET Framework class libraries, which, by default, are automatically referenced by the Microsoft C# compiler. Note that C# itself does not have a separate runtime library. Instead, the .NET Framework *is* the runtime library of C#.

www.it-ebooks.info

BRAD ABRAMS It is interesting to note that `Console.WriteLine()` is simply a shortcut for `Console.Out.WriteLine`. `Console.Out` is a property that returns an implementation of the `System.IO.TextWriter` base class designed to output to the console. The preceding example could be written equally correctly as follows:

```
using System;
class Hello
{
    static void Main() {
        Console.Out.WriteLine("Hello, World");
    }
}
```

Early in the design of the framework, we kept a careful eye on exactly how this section of the C# language specification would have to be written as a bellwether of the complexity of the language. We opted to add the convenience overload on `Console` to make "Hello, World" that much easier to write. By all accounts, it seems to have paid off. In fact, today you find almost no calls to `Console.Out.WriteLine()`.

1.2 Program Structure

The key organizational concepts in C# are ***programs***, ***namespaces***, ***types***, ***members***, and ***assemblies***. C# programs consist of one or more source files. Programs declare types, which contain members and can be organized into namespaces. Classes and interfaces are examples of types. Fields, methods, properties, and events are examples of members. When C# programs are compiled, they are physically packaged into assemblies. Assemblies typically have the file extension `.exe` or `.dll`, depending on whether they implement ***applications*** or ***libraries***.

The example

```
using System;

namespace Acme.Collections
{
    public class Stack
    {
        Entry top;

        public void Push(object data) {
            top = new Entry(top, data);
        }

        public object Pop() {
            if (top == null) throw new InvalidOperationException();
            object result = top.data;
```


www.it-ebooks.info

```
            top = top.next;
            return result;
        }

        class Entry
        {
            public Entry next;
            public object data;

            public Entry(Entry next, object data) {
                this.next = next;
                this.data = data;
            }
        }
    }
}
```

declares a class named `Stack` in a namespace called `Acme.Collections`. The fully qualified name of this class is `Acme.Collections.Stack`. The class contains several members: a field named `top`, two methods named `Push` and `Pop`, and a nested class named `Entry`. The `Entry` class further contains three members: a field named `next`, a field named `data`, and a constructor. Assuming that the source code of the example is stored in the file `acme.cs`, the command line

```
csc /t:library acme.cs
```

compiles the example as a library (code without a `Main` entry point) and produces an assembly named `acme.dll`.

Assemblies contain executable code in the form of ***Intermediate Language*** (IL) instructions, and symbolic information in the form of ***metadata***. Before it is executed, the IL code in an assembly is automatically converted to processor-specific code by the Just-In-Time (JIT) compiler of .NET Common Language Runtime.

Because an assembly is a self-describing unit of functionality containing both code and metadata, there is no need for `#include` directives and header files in C#. The public types and members contained in a particular assembly are made available in a C# program simply by referencing that assembly when compiling the program. For example, this program uses the `Acme.Collections.Stack` class from the `acme.dll` assembly:

```
using System;
using Acme.Collections;

class Test
{
    static void Main() {
        Stack s = new Stack();
        s.Push(1);
        s.Push(10);
        s.Push(100);
```


www.it-ebooks.info

```
            Console.WriteLine(s.Pop());
            Console.WriteLine(s.Pop());
            Console.WriteLine(s.Pop());
        }
    }
```

If the program is stored in the file `test.cs`, when `test.cs` is compiled, the `acme.dll` assembly can be referenced using the compiler's `/r` option:

```
csc /r:acme.dll test.cs
```

This creates an executable assembly named `test.exe`, which, when run, produces the following output:

```
100
10
1
```

C# permits the source text of a program to be stored in several source files. When a multi-file C# program is compiled, all of the source files are processed together, and the source files can freely reference one another—conceptually, it is as if all the source files were concatenated into one large file before being processed. Forward declarations are never needed in C# because, with very few exceptions, declaration order is insignificant. C# does not limit a source file to declaring only one public type nor does it require the name of the source file to match a type declared in the source file.

ERIC LIPPERT This is unlike the Java language. Also, the fact that the declaration order is insignificant in C# is unlike the C++ language.

CHRIS SELLS Notice in the previous example the `using Acme.Collections` statement, which looks like a C-style `#include` directive, but isn't. Instead, it's merely a naming convenience so that when the compiler encounters the `Stack`, it has a set of namespaces in which to look for the class. The compiler would take the same action if this example used the fully qualified name:

```
Acme.Collections.Stack s = new Acme.Collections.Stack();
```

1.3 Types and Variables

There are two kinds of types in C#: ***value types*** and ***reference types***. Variables of value types directly contain their data, whereas variables of reference types store references to their data, the latter being known as objects. With reference types, it is possible for two

www.it-ebooks.info

variables to reference the same object and, therefore, possible for operations on one variable to affect the object referenced by the other variable. With value types, the variables each have their own copy of the data, and it is not possible for operations on one to affect the other (except in the case of `ref` and `out` parameter variables).

JON SKEET The choice of the word "reference" for reference types is perhaps unfortunate. It has led to huge amounts of confusion (or at least miscommunication) when considering the difference between pass-by-reference and pass-by-value semantics for parameter passing.

The difference between value types and reference types is possibly the most important point to teach C# beginners: Until that point is understood, almost nothing else makes sense.

ERIC LIPPERT Probably the most common misconception about value types is that they are "stored on the stack," whereas reference types are "stored on the heap." First, that behavior is an implementation detail of the runtime, not a fact about the language. Second, it explains nothing to the novice. Third, it's false: Yes, the data associated with an instance of a reference type is stored on the heap, but that data can include instances of value types and, therefore, value types are also stored on the heap sometimes. Fourth, if the difference between value and reference types was their storage details, then the CLR team would have called them "stack types" and "heap types." The real difference is that value types are copied by value, and reference types are copied by reference; how the runtime allocates storage to implement the lifetime rules is not important in the vast majority of mainline programming scenarios.

BILL WAGNER C# forces you to make the important decision of value semantics versus reference semantics for your types. Developers using your type do not get to make that decision on each usage (as they do in C++). You need to think about the usage patterns for your types and make a careful decision between these two kinds of types.

VLADIMIR RESHETNIKOV C# also supports unsafe pointer types, which are described at the end of this specification. They are called "unsafe" because their negligent use can break the type safety in a way that cannot be caught by the compiler.

www.it-ebooks.info

C#'s value types are further divided into ***simple types***, ***enum types***, ***struct types***, and ***nullable types***. C#'s reference types are further divided into ***class types***, ***interface types***, ***array types***, and ***delegate types***.

The following table provides an overview of C#'s type system.

Category		Description
Value types	Simple types	Signed integral: `sbyte`, `short`, `int`, `long`
		Unsigned integral: `byte`, `ushort`, `uint`, `ulong`
		Unicode characters: `char`
		IEEE floating point: `float`, `double`
		High-precision decimal: `decimal`
		Boolean: `bool`
	Enum types	User-defined types of the form `enum E {...}`
	Struct types	User-defined types of the form `struct S {...}`
	Nullable types	Extensions of all other value types with a `null` value
Reference types	Class types	Ultimate base class of all other types: `object`
		Unicode strings: `string`
		User-defined types of the form `class C {...}`
	Interface types	User-defined types of the form `interface I {...}`
	Array types	Single- and multi-dimensional; for example, `int[]` and `int[,]`
	Delegate types	User-defined types of the form e.g. `delegate int D(...)`

The eight integral types provide support for 8-bit, 16-bit, 32-bit, and 64-bit values in signed or unsigned form.

www.it-ebooks.info

JON SKEET Hooray for `byte` being an unsigned type! The fact that in Java a byte is signed (and with no unsigned equivalent) makes a lot of bit-twiddling pointlessly error-prone.

It's quite possible that we should all be using `uint` a lot more than we do, mind you: I'm sure many developers reach for `int` by default when they want an integer type. The framework designers also fall into this category, of course: Why should `String.Length` be signed?

ERIC LIPPERT The answer to Jon's question is that the framework is designed to work well with the Common Language Specification (CLS). The CLS defines a set of basic language features that all CLS-compliant languages are expected to be able to consume; unsigned integers are not in the CLS subset.

The two floating point types, `float` and `double`, are represented using the 32-bit single-precision and 64-bit double-precision IEEE 754 formats.

The `decimal` type is a 128-bit data type suitable for financial and monetary calculations.

JON SKEET These two paragraphs imply that `decimal` isn't a floating point type. It is—it's just a floating *decimal* point type, whereas `float` and `double` are floating *binary* point types.

C#'s `bool` type is used to represent boolean values—values that are either `true` or `false`.

Character and string processing in C# uses Unicode encoding. The `char` type represents a UTF-16 code unit, and the `string` type represents a sequence of UTF-16 code units.

The following table summarizes C#'s numeric types.

Category	Bits	Type	Range/Precision
Signed integral	8	`sbyte`	–128...127
	16	`short`	–32,768...32,767
	32	`int`	–2,147,483,648...2,147,483,647
	64	`long`	–9,223,372,036,854,775,808...9,223,372,036,854,775,807

Continued

www.it-ebooks.info

Category	Bits	Type	Range/Precision
Unsigned integral	8	`byte`	0...255
	16	`ushort`	0...65,535
	32	`uint`	0...4,294,967,295
	64	`ulong`	0...18,446,744,073,709,551,615
Floating point	32	`float`	1.5×10^{-45} to 3.4×10^{38}, 7-digit precision
	64	`double`	5.0×10^{-324} to 1.7×10^{308}, 15-digit precision
Decimal	128	`decimal`	1.0×10^{-28} to 7.9×10^{28}, 28-digit precision

CHRISTIAN NAGEL One of the problems we had with C++ on different platforms is that the standard doesn't define the number of bits used with `short`, `int`, and `long`. The standard defines only `short <= int <= long`, which results in different sizes on 16-, 32-, and 64-bit platforms. With C#, the length of numeric types is clearly defined, no matter which platform is used.

C# programs use ***type declarations*** to create new types. A type declaration specifies the name and the members of the new type. Five of C#'s categories of types are user-definable: class types, struct types, interface types, enum types, and delegate types.

A class type defines a data structure that contains data members (fields) and function members (methods, properties, and others). Class types support single inheritance and polymorphism, mechanisms whereby derived classes can extend and specialize base classes.

ERIC LIPPERT Choosing to support single rather than multiple inheritance on classes eliminates in one stroke many of the complicated corner cases found in multiple inheritance languages.

A struct type is similar to a class type in that it represents a structure with data members and function members. However, unlike classes, structs are value types and do not require heap allocation. Struct types do not support user-specified inheritance, and all struct types implicitly inherit from type `object`.

www.it-ebooks.info

VLADIMIR RESHETNIKOV Structs inherit from `object` indirectly. Their implicit direct base class is `System.ValueType`, which in turn directly inherits from `object`.

An interface type defines a contract as a named set of public function members. A class or struct that implements an interface must provide implementations of the interface's function members. An interface may inherit from multiple base interfaces, and a class or struct may implement multiple interfaces.

A delegate type represents references to methods with a particular parameter list and return type. Delegates make it possible to treat methods as entities that can be assigned to variables and passed as parameters. Delegates are similar to the concept of function pointers found in some other languages, but unlike function pointers, delegates are object-oriented and type-safe.

Class, struct, interface, and delegate types all support generics, whereby they can be parameterized with other types.

An enum type is a distinct type with named constants. Every enum type has an underlying type, which must be one of the eight integral types. The set of values of an enum type is the same as the set of values of the underlying type.

VLADIMIR RESHETNIKOV Enum types cannot have type parameters in their declarations. Even so, they can be generic if nested within a generic class or struct type. Moreover, C# supports pointers to generic enum types in unsafe code.

Sometimes enum types are called "enumeration types" in this specification. These two names are completely interchangeable.

C# supports single- and multi-dimensional arrays of any type. Unlike the types listed above, array types do not have to be declared before they can be used. Instead, array types are constructed by following a type name with square brackets. For example, `int[]` is a single-dimensional array of `int`, `int[,]` is a two-dimensional array of `int`, and `int[][]` is a single-dimensional array of single-dimensional arrays of `int`.

Nullable types also do not have to be declared before they can be used. For each non-nullable value type `T` there is a corresponding nullable type `T?`, which can hold an additional value `null`. For instance, `int?` is a type that can hold any 32 bit integer or the value `null`.

CHRISTIAN NAGEL `T?` is the C# shorthand notation for the `Nullable<T>` structure.

www.it-ebooks.info

> **ERIC LIPPERT** In C# 1.0, we had nullable reference types and non-nullable value types. In C# 2.0, we added nullable value types. But there are no non-nullable reference types. If we had to do it all over again, we probably would bake nullability and non-nullability into the type system from day one. Unfortunately, non-nullable reference types are difficult to add to an existing type system that wasn't designed for them. We get feature requests for non-nullable reference types all the time; it would be a great feature. However, code contracts go a long way toward solving the problems solved by non-nullable reference types; consider using them if you want to enforce non-nullability in your programs. If this subject interests you, you might also want to check out Spec#, a Microsoft Research version of C# that does support non-nullable reference types.

C#'s type system is unified such that a value of any type can be treated as an object. Every type in C# directly or indirectly derives from the `object` class type, and `object` is the ultimate base class of all types. Values of reference types are treated as objects simply by viewing the values as type `object`. Values of value types are treated as objects by performing ***boxing*** and ***unboxing*** operations. In the following example, an `int` value is converted to `object` and back again to `int`.

```
using System;

class Test
{
    static void Main() {
        int i = 123;
        object o = i;           // Boxing
        int j = (int)o;         // Unboxing
    }
}
```

When a value of a value type is converted to type `object`, an object instance, also called a "box," is allocated to hold the value, and the value is copied into that box. Conversely, when an `object` reference is cast to a value type, a check is made that the referenced object is a box of the correct value type, and, if the check succeeds, the value in the box is copied out.

C#'s unified type system effectively means that value types can become objects "on demand." Because of the unification, general-purpose libraries that use type `object` can be used with both reference types and value types.

There are several kinds of ***variables*** in C#, including fields, array elements, local variables, and parameters. Variables represent storage locations, and every variable has a type that determines what values can be stored in the variable, as shown by the following table.

www.it-ebooks.info

Type of Variable	Possible Contents
Non-nullable value type	A value of that exact type
Nullable value type	A null value or a value of that exact type
`object`	A null reference, a reference to an object of any reference type, or a reference to a boxed value of any value type
Class type	A null reference, a reference to an instance of that class type, or a reference to an instance of a class derived from that class type
Interface type	A null reference, a reference to an instance of a class type that implements that interface type, or a reference to a boxed value of a value type that implements that interface type
Array type	A null reference, a reference to an instance of that array type, or a reference to an instance of a compatible array type
Delegate type	A null reference or a reference to an instance of that delegate type

1.4 Expressions

Expressions are constructed from ***operands*** and ***operators***. The operators of an expression indicate which operations to apply to the operands. Examples of operators include `+`, `-`, `*`, `/`, and `new`. Examples of operands include literals, fields, local variables, and expressions.

When an expression contains multiple operators, the ***precedence*** of the operators controls the order in which the individual operators are evaluated. For example, the expression `x + y * z` is evaluated as `x + (y * z)` because the `*` operator has higher precedence than the `+` operator.

> **ERIC LIPPERT** Precedence controls the order in which the operators are executed, but not the order in which the operands are evaluated. Operands are evaluated from left to right, period. In the preceding example, `x` would be evaluated, then `y`, then `z`, then the multiplication would be performed, and then the addition. The evaluation of operand `x` happens before that of `y` because `x` is to the left of `y`; the evaluation of the multiplication happens before the addition because the multiplication has higher precedence.

Most operators can be ***overloaded***. Operator overloading permits user-defined operator implementations to be specified for operations where one or both of the operands are of a user-defined class or struct type.

www.it-ebooks.info

The following table summarizes C#'s operators, listing the operator categories in order of precedence from highest to lowest. Operators in the same category have equal precedence.

Category	Expression	Description
Primary	`x.m`	Member access
	`x(...)`	Method and delegate invocation
	`x[...]`	Array and indexer access
	`x++`	Post-increment
	`x--`	Post-decrement
	`new T(...)`	Object and delegate creation
	`new T(...){...}`	Object creation with initializer
	`new {...}`	Anonymous object initializer
	`new T[...]`	Array creation
	`typeof(T)`	Obtain `System.Type` object for `T`
	`checked(x)`	Evaluate expression in checked context
	`unchecked(x)`	Evaluate expression in unchecked context
	`default(T)`	Obtain default value of type `T`
	`delegate {...}`	Anonymous function (anonymous method)
Unary	`+x`	Identity
	`-x`	Negation
	`!x`	Logical negation
	`~x`	Bitwise negation
	`++x`	Pre-increment
	`--x`	Pre-decrement
	`(T)x`	Explicitly convert `x` to type `T`

www.it-ebooks.info

Category	Expression	Description
Multiplicative	`x * y`	Multiplication
	`x / y`	Division
	`x % y`	Remainder
Additive	`x + y`	Addition, string concatenation, delegate combination
	`x - y`	Subtraction, delegate removal
Shift	`x << y`	Shift left
	`x >> y`	Shift right
Relational and type testing	`x < y`	Less than
	`x > y`	Greater than
	`x <= y`	Less than or equal
	`x >= y`	Greater than or equal
	`x is T`	Return `true` if `x` is a `T`, `false` otherwise
	`x as T`	Return `x` typed as `T`, or `null` if `x` is not a `T`
Equality	`x == y`	Equal
	`x != y`	Not equal
Logical AND	`x & y`	Integer bitwise AND, boolean logical AND
Logical XOR	`x ^ y`	Integer bitwise XOR, boolean logical XOR
Logical OR	`x | y`	Integer bitwise OR, boolean logical OR
Conditional AND	`x && y`	Evaluates `y` only if `x` is `true`
Conditional OR	`x || y`	Evaluates `y` only if `x` is `false`
Null coalescing	`X ?? y`	Evaluates to `y` if `x` is `null`, to `x` otherwise
Conditional	`x ? y : z`	Evaluates `y` if `x` is `true`, `z` if `x` is `false`

Continued

www.it-ebooks.info

Category	Expression	Description
Assignment or anonymous function	`x = y`	Assignment
	`x` *op*`= y`	Compound assignment; supported operators are `*= /= %= += -= <<= >>= &= ^= |=`
	`(T x) => y`	Anonymous function (lambda expression)

ERIC LIPPERT It is often surprising to people that the lambda and anonymous method syntaxes are described as operators. They are unusual operators. More typically, you think of an operator as taking expressions as operands, not declarations of formal parameters. Syntactically, however, the lambda and anonymous method syntaxes are operators like any other.

1.5 Statements

The actions of a program are expressed using ***statements***. C# supports several kinds of statements, a number of which are defined in terms of embedded statements.

A ***block*** permits multiple statements to be written in contexts where a single statement is allowed. A block consists of a list of statements written between the delimiters `{` and `}`.

Declaration statements are used to declare local variables and constants.

Expression statements are used to evaluate expressions. Expressions that can be used as statements include method invocations, object allocations using the `new` operator, assignments using `=` and the compound assignment operators, and increment and decrement operations using the `++` and `--` operators.

Selection statements are used to select one of a number of possible statements for execution based on the value of some expression. In this group are the `if` and `switch` statements.

Iteration statements are used to repeatedly execute an embedded statement. In this group are the `while`, `do`, `for`, and `foreach` statements.

Jump statements are used to transfer control. In this group are the `break`, `continue`, `goto`, `throw`, `return`, and `yield` statements.

www.it-ebooks.info

The `try...catch` statement is used to catch exceptions that occur during execution of a block, and the `try...finally` statement is used to specify finalization code that is always executed, whether an exception occurred or not.

> **ERIC LIPPERT** This is a bit of a fib; of course, a `finally` block does not *always* execute. The code in the `try` block could go into an infinite loop, the exception could trigger a "fail fast" (which takes the process down without running any `finally` blocks), or someone could pull the power cord out of the wall.

The `checked` and `unchecked` statements are used to control the overflow checking context for integral-type arithmetic operations and conversions.

The `lock` statement is used to obtain the mutual-exclusion lock for a given object, execute a statement, and then release the lock.

The `using` statement is used to obtain a resource, execute a statement, and then dispose of that resource.

The following table lists C#'s statements and provides an example for each one.

Statement	Example

Local variable declaration	`static void Main() {`
	`  int a;`
	`  int b = 2, c = 3;`
	`  a = 1;`
	`  Console.WriteLine(a + b + c);`
	`}`

Local constant declaration	`static void Main() {`
	`  const float pi = 3.1415927f;`
	`  const int r = 25;`
	`  Console.WriteLine(pi * r * r);`
	`}`

Expression statement	`static void Main() {`
	`  int i;`
	`  i = 123;  // Expression statement`
	`  Console.WriteLine(i);  // Expression statement`
	`  i++;  // Expression statement`
	`  Console.WriteLine(i);  // Expression statement`
	`}`

Continued

www.it-ebooks.info

Statement	Example

if statement	`static void Main(string[] args) {`
	`  if (args.Length == 0) {`
	`  Console.WriteLine("No arguments");`
	`  }`
	`  else {`
	`  Console.WriteLine("One or more arguments");`
	`  }`
	`}`

switch statement	`static void Main(string[] args) {`
	`  int n = args.Length;`
	`  switch (n) {`
	`  case 0:`
	`  Console.WriteLine("No arguments");`
	`  break;`
	`  case 1:`
	`  Console.WriteLine("One argument");`
	`  break;`
	`  default:`
	`  Console.WriteLine("{0} arguments", n);`
	`  break;`
	`  }`
	`  }`
	`}`

while statement	`static void Main(string[] args) {`
	`  int i = 0;`
	`  while (i < args.Length) {`
	`  Console.WriteLine(args[i]);`
	`  i++;`
	`  }`
	`}`

do statement	`static void Main() {`
	`  string s;`
	`  do {`
	`  s = Console.ReadLine();`
	`  if (s != null) Console.WriteLine(s);`
	`  } while (s != null);`
	`}`

www.it-ebooks.info

Statement	Example

for statement	`static void Main(string[] args) {`
	`  for (int i = 0; i < args.Length; i++) {`
	`  Console.WriteLine(args[i]);`
	`  }`
	`}`

foreach statement	`static void Main(string[] args) {`
	`  foreach (string s in args) {`
	`  Console.WriteLine(s);`
	`  }`
	`}`

break statement	`static void Main() {`
	`  while (true) {`
	`  string s = Console.ReadLine();`
	`  if (s == null) break;`
	`  Console.WriteLine(s);`
	`  }`
	`}`

continue statement	`static void Main(string[] args) {`
	`  for (int i = 0; i < args.Length; i++) {`
	`  if (args[i].StartsWith("/")) continue;`
	`  Console.WriteLine(args[i]);`
	`  }`
	`}`

goto statement	`static void Main(string[] args) {`
	`  int i = 0;`
	`  goto check;`
	`  loop:`
	`  Console.WriteLine(args[i++]);`
	`  check:`
	`  if (i < args.Length) goto loop;`
	`}`

return statement	`static int Add(int a, int b) {`
	`  return a + b;`
	`}`
	`static void Main() {`
	`  Console.WriteLine(Add(1, 2));`
	`  return;`
	`}`

Continued

www.it-ebooks.info

Statement	Example

yield statement	`static IEnumerable<int> Range(int from, int to) {`
	`  for (int i = from; i < to; i++) {`
	`  yield return i;`
	`  }`
	`  yield break;`
	`}`
	`static void Main() {`
	`  foreach (int x in Range(-10,10)) {`
	`  Console.WriteLine(x);`
	`  }`
	`}`

throw and try statements	`static double Divide(double x, double y) {`
	`  if (y == 0) throw new DivideByZeroException();`
	`  return x / y;`
	`}`
	`static void Main(string[] args) {`
	`  try {`
	`  if (args.Length != 2) {`
	`  throw new Exception("Two numbers required");`
	`  }`
	`  double x = double.Parse(args[0]);`
	`  double y = double.Parse(args[1]);`
	`  Console.WriteLine(Divide(x, y));`
	`  }`
	`  catch (Exception e) {`
	`  Console.WriteLine(e.Message);`
	`  }`
	`  finally {`
	`  Console.WriteLine(“Good bye!”);`
	`  }`
	`}`

checked and unchecked statements	`static void Main() {`
	`  int i = int.MaxValue;`
	`  checked {`
	`  Console.WriteLine(i + 1);  // Exception`
	`  }`
	`  unchecked {`
	`  Console.WriteLine(i + 1);  // Overflow`
	`  }`
	`}`

www.it-ebooks.info

Statement	Example

`lock` statement	`class Account`
	`{`
	`  decimal balance;`
	`  public void Withdraw(decimal amount) {`
	`  lock (this) {`
	`  if (amount > balance) {`
	`  throw new Exception("Insufficient funds");`
	`  }`
	`  balance -= amount;`
	`  }`
	`  }`
	`}`

`using` statement	`static void Main() {`
	`  using (TextWriter w = File.CreateText("test.txt")) {`
	`  w.WriteLine("Line one");`
	`  w.WriteLine("Line two");`
	`  w.WriteLine("Line three");`
	`  }`
	`}`

1.6 Classes and Objects

Classes are the most fundamental of C#'s types. A class is a data structure that combines state (fields) and actions (methods and other function members) in a single unit. A class provides a definition for dynamically created ***instances*** of the class, also known as ***objects***. Classes support ***inheritance*** and ***polymorphism***, mechanisms whereby ***derived classes*** can extend and specialize ***base classes***.

New classes are created using class declarations. A class declaration starts with a header that specifies the attributes and modifiers of the class, the name of the class, the base class (if given), and the interfaces implemented by the class. The header is followed by the class body, which consists of a list of member declarations written between the delimiters { and }.

www.it-ebooks.info

The following is a declaration of a simple class named `Point`:

```
public class Point
{
    public int x, y;

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

Instances of classes are created using the `new` operator, which allocates memory for a new instance, invokes a constructor to initialize the instance, and returns a reference to the instance. The following statements create two `Point` objects and store references to those objects in two variables:

```
Point p1 = new Point(0, 0);
Point p2 = new Point(10, 20);
```

The memory occupied by an object is automatically reclaimed when the object is no longer in use. It is neither necessary nor possible to explicitly deallocate objects in C#.

1.6.1 Members

The members of a class are either ***static members*** or ***instance members***. Static members belong to classes, and instance members belong to objects (instances of classes).

> **ERIC LIPPERT** The term "static" was chosen because of its familiarity to users of similar languages, rather than because it is a particularly sensible or descriptive term for "shared by all instances of a class."

> **JON SKEET** I'd argue that "shared" (as used in Visual Basic) gives an incorrect impression, too. "Sharing" feels like something that requires one or more participants, whereas a static member doesn't require *any* instances of the type. I have the perfect term for this situation, but it's too late to change "static" to "associated-with-the-type-rather-than-with-any-specific-instance-of-the-type" (hyphens optional).

The following table provides an overview of the kinds of members a class can contain.

www.it-ebooks.info

Member	Description
Constants	Constant values associated with the class
Fields	Variables of the class
Methods	Computations and actions that can be performed by the class
Properties	Actions associated with reading and writing named properties of the class
Indexers	Actions associated with indexing instances of the class like an array
Events	Notifications that can be generated by the class
Operators	Conversions and expression operators supported by the class
Constructors	Actions required to initialize instances of the class or the class itself
Destructors	Actions to perform before instances of the class are permanently discarded
Types	Nested types declared by the class

1.6.2 Accessibility

Each member of a class has an associated accessibility, which controls the regions of program text that are able to access the member. The five possible forms of accessibility are summarized in the following table.

Accessibility	Meaning
`public`	Access not limited
`protected`	Access limited to this class or classes derived from this class
`internal`	Access limited to this program
`protected internal`	Access limited to this program or classes derived from this class
`private`	Access limited to this class

KRZYSZTOF CWALINA People need to be careful with the `public` keyword. `public` in C# is *not* equivalent to `public` in C++! In C++, it means "internal to my compilation unit." In C#, it means what `extern` meant in C++ (i.e., everybody can call it). This is a huge difference!

www.it-ebooks.info

CHRISTIAN NAGEL I would describe the internal access modifier as "access limited to this assembly" instead of "access limited to this program." If the internal access modifier is used within a DLL, the EXE referencing the DLL does not have access to it.

ERIC LIPPERT `protected internal` has proven to be a controversial and somewhat unfortunate choice. Many people using this feature incorrectly believe that `protected internal` means "access is limited to derived classes within this program." That is, they believe it means the *more* restrictive combination, when in fact it means the *less* restrictive combination. The way to remember this relationship is to remember that the "natural" state of a member is "private" and every accessibility modifier makes the accessibility domain *larger*.

Were a hypothetical future version of the C# language to provide a syntax for "the more restrictive combination of protected and internal," the question would then be which combination of keywords would have that meaning. I am holding out for either "proternal" or "intected," but I suspect I will have to live with disappointment.

CHRISTIAN NAGEL C# defines `protected internal` to limit access to this assembly *or* classes derived from this class. The CLR also allows limiting access to this assembly *and* classes derived from this class. C++/CLI offers this CLR feature with the `public private` access modifier (or `private public`—the order is not relevant). Realistically, this access modifier is rarely used.

1.6.3 Type Parameters

A class definition may specify a set of type parameters by following the class name with angle brackets enclosing a list of type parameter names. The type parameters can then be used in the body of the class declarations to define the members of the class. In the following example, the type parameters of `Pair` are `TFirst` and `TSecond`:

```
public class Pair<TFirst, TSecond>
{
    public TFirst First;

    public TSecond Second;
}
```

A class type that is declared to take type parameters is called a generic class type. Struct, interface, and delegate types can also be generic.

www.it-ebooks.info

> **ERIC LIPPERT** If you need a pair, triple, and so on, the generic "tuple" types defined in the CLR 4 version of the framework are handy types.

When the generic class is used, type arguments must be provided for each of the type parameters:

```
Pair<int,string> pair = new Pair<int,string> { First = 1, Second = "two" };
int i = pair.First;     // TFirst is int
string s = pair.Second; // TSecond is string
```

A generic type with type arguments provided, like `Pair<int,string>` above, is called a constructed type.

1.6.4 Base Classes

A class declaration may specify a base class by following the class name and type parameters with a colon and the name of the base class. Omitting a base class specification is the same as deriving from type `object`. In the following example, the base class of `Point3D` is `Point`, and the base class of `Point` is `object`:

```
public class Point
{
    public int x, y;

    public Point(int x, int y)
    {
        this.x = x;
        this.y = y;
    }
}

public class Point3D : Point
{
    public int z;

    public Point3D(int x, int y, int z): base(x, y)
    {
        this.z = z;
    }
}
```

A class inherits the members of its base class. Inheritance means that a class implicitly contains all members of its base class, except for the instance and static constructors, and the destructors of the base class. A derived class can add new members to those it inherits, but it cannot remove the definition of an inherited member. In the previous example, `Point3D`

www.it-ebooks.info

inherits the `x` and `y` fields from `Point`, and every `Point3D` instance contains three fields, `x`, `y`, and `z`.

> **JESSE LIBERTY** There is nothing more important to understand about C# than inheritance and polymorphism. These concepts are the heart of the language and the soul of object-oriented programming. Read this section until it makes sense, or ask for help or supplement it with additional reading, but do not skip over it—these issues are the sine qua non of C#.

An implicit conversion exists from a class type to any of its base class types. Therefore, a variable of a class type can reference an instance of that class or an instance of any derived class. For example, given the previous class declarations, a variable of type `Point` can reference either a `Point` or a `Point3D`:

```
Point a = new Point(10, 20);
Point b = new Point3D(10, 20, 30);
```

1.6.5 Fields

A field is a variable that is associated with a class or with an instance of a class.

A field declared with the `static` modifier defines a ***static field***. A static field identifies exactly one storage location. No matter how many instances of a class are created, there is only ever one copy of a static field.

> **ERIC LIPPERT** Static fields are per *constructed* type for a generic type. That is, if you have a
>
> ```
> class Stack<T> {
>         public readonly static Stack<T> empty = whatever; ...
> }
> ```
>
> then `Stack<int>.empty` is a different field than `Stack<string>.empty`.

A field declared without the `static` modifier defines an ***instance field***. Every instance of a class contains a separate copy of all the instance fields of that class.

In the following example, each instance of the `Color` class has a separate copy of the `r`, `g`, and `b` instance fields, but there is only one copy of the `Black`, `White`, `Red`, `Green`, and `Blue` static fields:

www.it-ebooks.info

```
public class Color
{
    public static readonly Color Black = new Color(0, 0, 0);
    public static readonly Color White = new Color(255, 255, 255);
    public static readonly Color Red = new Color(255, 0, 0);
    public static readonly Color Green = new Color(0, 255, 0);
    public static readonly Color Blue = new Color(0, 0, 255);

    private byte r, g, b;

    public Color(byte r, byte g, byte b)
    {
        this.r = r;
        this.g = g;
        this.b = b;
    }
}
```

As shown in the previous example, ***read-only fields*** may be declared with a `readonly` modifier. Assignment to a `readonly` field can occur only as part of the field's declaration or in a constructor in the same class.

BRAD ABRAMS `readonly` protects the location of the field from being changed outside the type's constructor, but does not protect the value at that location. For example, consider the following type:

```
public class Names
{
    public static readonly StringBuilder FirstBorn = new StringBuilder("Joe");
    public static readonly StringBuilder SecondBorn = new StringBuilder("Sue");
}
```

Outside of the constructor, directly changing the `FirstBorn` instance results in a compiler error:

```
Names.FirstBorn = new StringBuilder("Biff");
// Compile error
```

However, I am able to accomplish exactly the same results by modifying the `StringBuilder` instance:

```
Names.FirstBorn.Remove(0,6).Append("Biff");
Console.WriteLine(Names.FirstBorn);       // Outputs "Biff"
```

It is for this reason that we strongly recommend that read-only fields be limited to immutable types. Immutable types do not have any publicly exposed setters, such as `int`, `double`, or `String`.

www.it-ebooks.info

BILL WAGNER Several well-known design patterns make use of the read-only fields of mutable types. The Adapter, Decorator, Façade, and Proxy patterns are the most obvious examples. When you are creating a larger structure by composing smaller structures, you will often express instances of those smaller structures using read-only fields. A read-only field of a mutable type should indicate that one of these structural patterns is being used.

1.6.6 Methods

A ***method*** is a member that implements a computation or action that can be performed by an object or class. ***Static methods*** are accessed through the class. ***Instance methods*** are accessed through instances of the class.

Methods have a (possibly empty) list of ***parameters***, which represent values or variable references passed to the method, and a ***return type***, which specifies the type of the value computed and returned by the method. A method's return type is `void` if it does not return a value.

Like types, methods may also have a set of type parameters, for which type arguments must be specified when the method is called. Unlike types, the type arguments can often be inferred from the arguments of a method call and need not be explicitly given.

The ***signature*** of a method must be unique in the class in which the method is declared. The signature of a method consists of the name of the method, the number of type parameters, and the number, modifiers, and types of its parameters. The signature of a method does not include the return type.

ERIC LIPPERT An unfortunate consequence of generic types is that a constructed type may potentially have two methods with identical signatures. For example, `class C<T> { void M(T t){} void M(int t){} ...}` is perfectly legal, but `C<int>` has two methods `M` with identical signatures. As we'll see later on, this possibility leads to some interesting scenarios involving overload resolution and explicit interface implementations. A good guideline: Don't create a generic type that can create ambiguities under construction in this way; such types are extremely confusing and can produce unexpected behaviors.

www.it-ebooks.info

1.6.6.1 *Parameters*

Parameters are used to pass values or variable references to methods. The parameters of a method get their actual values from the ***arguments*** that are specified when the method is invoked. There are four kinds of parameters: value parameters, reference parameters, output parameters, and parameter arrays.

A ***value parameter*** is used for input parameter passing. A value parameter corresponds to a local variable that gets its initial value from the argument that was passed for the parameter. Modifications to a value parameter do not affect the argument that was passed for the parameter.

> **BILL WAGNER** The statement that modifications to value parameters do not affect the argument might be misleading because mutator methods may change the contents of a parameter of reference type. The value parameter does not change, but the contents of the referred-to object do.

Value parameters can be optional, by specifying a default value so that corresponding arguments can be omitted.

A ***reference parameter*** is used for both input and output parameter passing. The argument passed for a reference parameter must be a variable, and during execution of the method, the reference parameter represents the same storage location as the argument variable. A reference parameter is declared with the `ref` modifier. The following example shows the use of `ref` parameters.

```
using System;

class Test
{
    static void Swap(ref int x, ref int y) {
        int temp = x;
        x = y;
        y = temp;
    }

    static void Main() {
        int i = 1, j = 2;
        Swap(ref i, ref j);
        Console.WriteLine("{0} {1}", i, j); // Outputs "2 1"
    }
}
```


www.it-ebooks.info

ERIC LIPPERT This syntax should help clear up the confusion between the two things both called "passing by reference." Reference types are called this name in C# because they are "passed by reference"; you pass an object instance to a method, and the method gets a reference to that object instance. Some other code might also be holding on to a reference to the same object.

Reference parameters are a slightly different form of "passing by reference." In this case, the reference is to the variable itself, not to some object instance. If that variable happens to contain a value type (as shown in the previous example), that's perfectly legal. The value is not being passed by reference, but rather the variable that holds it is.

A good way to think about reference parameters is that the reference parameter becomes an *alias* for the variable passed as the argument. In the preceding example, `x` and `i` are essentially *the same variable.* They refer to the same storage location.

An ***output parameter*** is used for output parameter passing. An output parameter is similar to a reference parameter except that the initial value of the caller-provided argument is unimportant. An output parameter is declared with the `out` modifier. The following example shows the use of `out` parameters.

```
using System;

class Test
{
    static void Divide(int x, int y, out int result, out int remainder) {
        result = x / y;
        remainder = x % y;
    }

    static void Main() {
        int res, rem;
        Divide(10, 3, out res, out rem);
        Console.WriteLine("{0} {1}", res, rem);     // Outputs "3 1"
    }
}
```

ERIC LIPPERT The CLR directly supports only `ref` parameters. An `out` parameter is represented in metadata as a `ref` parameter with a special attribute on it indicating to the C# compiler that this `ref` parameter ought to be treated as an `out` parameter. This explains why it is not legal to have two methods that differ solely in "out/ref-ness"; from the CLR's perspective, they would be two identical methods.

www.it-ebooks.info

A ***parameter array*** permits a variable number of arguments to be passed to a method. A parameter array is declared with the `params` modifier. Only the last parameter of a method can be a parameter array, and the type of a parameter array must be a single-dimensional array type. The `Write` and `WriteLine` methods of the `System.Console` class are good examples of parameter array usage. They are declared as follows.

```
public class Console
{
    public static void Write(string fmt, params object[] args) {...}

    public static void WriteLine(string fmt, params object[] args) {...}

    ...
}
```

Within a method that uses a parameter array, the parameter array behaves exactly like a regular parameter of an array type. However, in an invocation of a method with a parameter array, it is possible to pass either a single argument of the parameter array type or any number of arguments of the element type of the parameter array. In the latter case, an array instance is automatically created and initialized with the given arguments. This example

```
Console.WriteLine("x={0} y={1} z={2}", x, y, z);
```

is equivalent to writing the following.

```
string s = "x={0} y={1} z={2}";
object[] args = new object[3];
args[0] = x;
args[1] = y;
args[2] = z;
Console.WriteLine(s, args);
```

BRAD ABRAMS You may recognize the similarity between `params` and the C programming language's `varargs` concept. In keeping with our goal of making C# very simple to understand, the `params` modifier does not require a special calling convention or special library support. As such, it has proven to be much less prone to error than `varargs`.

Note, however, that the C# model does create an extra object allocation (the containing array) implicitly on each call. This is rarely a problem, but in inner-loop type scenarios where it could get inefficient, we suggest providing overloads for the mainstream cases and using the `params` overload for only the edge cases. An example is the `StringBuilder.AppendFormat()` family of overloads:

```
public StringBuilder AppendFormat(string format, object arg0);
public StringBuilder AppendFormat(string format, object arg0, object arg1);
public StringBuilder AppendFormat(string format, object arg0, object arg1, object arg2);
public StringBuilder AppendFormat(string format, params object[] args);
```


www.it-ebooks.info

CHRIS SELLS One nice side effect of the fact that `params` is really just an optional shortcut is that I don't have to write something crazy like the following:

```
static object[] GetArgs() { ... }

static void Main() {
        object[] args = GetArgs();
        object x = args[0];
        object y = args[1];
        object z = args[2];
        Console.WriteLine("x={0} y={1} z={2}", x, y, z);
}
```

Here I'm calling the method and cracking the parameters out just so the compiler can create an array around them again. Of course, I should really just write this:

```
static object[] GetArgs() { ... }

static void Main() {
        Console.WriteLine("x={0} y={1} z={2}", GetArgs());
}
```

However, you'll find fewer and fewer methods that return arrays in .NET these days, as most folks prefer using `IEnumerable<T>` for its flexibility. This means you'll probably be writing code like so:

```
static IEnumerable<object> GetArgs() { ... }

static void Main() {
        Console.WriteLine("x={0} y={1} z={2}", GetArgs().ToArray());
}
```

It would be handy if `params` "understood" `IEnumerable` directly. Maybe next time.

1.6.6.2 *Method Body and Local Variables*

A method's body specifies the statements to execute when the method is invoked.

A method body can declare variables that are specific to the invocation of the method. Such variables are called ***local variables***. A local variable declaration specifies a type name, a variable name, and possibly an initial value. The following example declares a local variable `i` with an initial value of zero and a local variable `j` with no initial value.

www.it-ebooks.info

```
using System;

class Squares
{
    static void Main() {
        int i = 0;
        int j;
        while (i < 10) {
            j = i * i;
            Console.WriteLine("{0} x {0} = {1}", i, j);
            i = i + 1;
        }
    }
}
```

C# requires a local variable to be ***definitely assigned*** before its value can be obtained. For example, if the declaration of the previous `i` did not include an initial value, the compiler would report an error for the subsequent usages of `i` because `i` would not be definitely assigned at those points in the program.

A method can use `return` statements to return control to its caller. In a method returning `void`, `return` statements cannot specify an expression. In a method returning non-`void`, `return` statements must include an expression that computes the return value.

1.6.6.3 *Static and Instance Methods*

A method declared with a `static` modifier is a ***static method***. A static method does not operate on a specific instance and can only directly access static members.

> **ERIC LIPPERT** It is, of course, perfectly legal for a static method to access instance members should it happen to have an instance handy.

A method declared without a `static` modifier is an ***instance method***. An instance method operates on a specific instance and can access both static and instance members. The instance on which an instance method was invoked can be explicitly accessed as `this`. It is an error to refer to `this` in a static method.

The following `Entity` class has both static and instance members.

```
class Entity
{
    static int nextSerialNo;

    int serialNo;

    public Entity()
    {
```


www.it-ebooks.info

```
            serialNo = nextSerialNo++;
        }
        public int GetSerialNo()
        {
            return serialNo;
        }

        public static int GetNextSerialNo()
        {
            return nextSerialNo;
        }

        public static void SetNextSerialNo(int value)
        {
            nextSerialNo = value;
        }
    }
```

Each `Entity` instance contains a serial number (and presumably some other information that is not shown here). The `Entity` constructor (which is like an instance method) initializes the new instance with the next available serial number. Because the constructor is an instance member, it is permitted to access both the `serialNo` instance field and the `nextSerialNo` static field.

The `GetNextSerialNo` and `SetNextSerialNo` static methods can access the `nextSerialNo` static field, but it would be an error for them to directly access the `serialNo` instance field.

The following example shows the use of the `Entity` class.

```
using System;

class Test
{
    static void Main() {
        Entity.SetNextSerialNo(1000);

        Entity e1 = new Entity();
        Entity e2 = new Entity();

        Console.WriteLine(e1.GetSerialNo());            // Outputs "1000"
        Console.WriteLine(e2.GetSerialNo());            // Outputs "1001"
        Console.WriteLine(Entity.GetNextSerialNo());    // Outputs "1002"
    }
}
```

Note that the `SetNextSerialNo` and `GetNextSerialNo` static methods are invoked on the class, whereas the `GetSerialNo` instance method is invoked on instances of the class.

www.it-ebooks.info

1.6.6.4 *Virtual, Override, and Abstract Methods*

When an instance method declaration includes a `virtual` modifier, the method is said to be a ***virtual method***. When no `virtual` modifier is present, the method is said to be a ***nonvirtual method***.

When a virtual method is invoked, the ***runtime type*** of the instance for which that invocation takes place determines the actual method implementation to invoke. In a nonvirtual method invocation, the ***compile-time type*** of the instance is the determining factor.

A virtual method can be ***overridden*** in a derived class. When an instance method declaration includes an `override` modifier, the method overrides an inherited virtual method with the same signature. Whereas a virtual method declaration *introduces* a new method, an override method declaration *specializes* an existing inherited virtual method by providing a new implementation of that method.

> **ERIC LIPPERT** A subtle point here is that an overridden virtual method is still considered to be a method of the class that introduced it, and not a method of the class that overrides it. The overload resolution rules in some cases prefer members of more derived types to those in base types; overriding a method does not "move" where that method belongs in this hierarchy.
>
> At the very beginning of this section, we noted that C# was designed with versioning in mind. This is one of those features that helps prevent "brittle base-class syndrome" from causing versioning problems.

An ***abstract*** method is a virtual method with no implementation. An abstract method is declared with the `abstract` modifier and is permitted only in a class that is also declared `abstract`. An abstract method must be overridden in every non-abstract derived class.

The following example declares an abstract class, `Expression`, which represents an expression tree node, and three derived classes, `Constant`, `VariableReference`, and `Operation`, which implement expression tree nodes for constants, variable references, and arithmetic operations. (This is similar to, but not to be confused with, the expression tree types introduced in §4.6).

```
using System;
using System.Collections;

public abstract class Expression
{
    public abstract double Evaluate(Hashtable vars);
}
```


www.it-ebooks.info

```
public class Constant: Expression
{
    double value;

    public Constant(double value) {
        this.value = value;
    }

    public override double Evaluate(Hashtable vars) {
        return value;
    }
}
public class VariableReference: Expression
{
    string name;

    public VariableReference(string name) {
        this.name = name;
    }

    public override double Evaluate(Hashtable vars) {
        object value = vars[name];
        if (value == null) {
            throw new Exception("Unknown variable: " + name);
        }
        return Convert.ToDouble(value);
    }
}

public class Operation: Expression
{
    Expression left;
    char op;
    Expression right;

    public Operation(Expression left, char op, Expression right) {
        this.left = left;
        this.op = op;
        this.right = right;
    }

    public override double Evaluate(Hashtable vars) {
        double x = left.Evaluate(vars);
        double y = right.Evaluate(vars);
        switch (op) {
            case '+': return x + y;
            case '-': return x - y;
            case '*': return x * y;
            case '/': return x / y;
        }
        throw new Exception("Unknown operator");
    }
}
```


www.it-ebooks.info

The previous four classes can be used to model arithmetic expressions. For example, using instances of these classes, the expression `x + 3` can be represented as follows.

```
Expression e = new Operation(
    new VariableReference("x"),
    '+',
    new Constant(3));
```

The `Evaluate` method of an `Expression` instance is invoked to evaluate the given expression and produce a `double` value. The method takes as an argument a `Hashtable` that contains variable names (as keys of the entries) and values (as values of the entries). The `Evaluate` method is a virtual abstract method, meaning that non-abstract derived classes must override it to provide an actual implementation.

A `Constant`'s implementation of `Evaluate` simply returns the stored constant. A `VariableReference`'s implementation looks up the variable name in the hashtable and returns the resulting value. An `Operation`'s implementation first evaluates the left and right operands (by recursively invoking their `Evaluate` methods) and then performs the given arithmetic operation.

The following program uses the `Expression` classes to evaluate the expression `x * (y + 2)` for different values of `x` and `y`.

```
using System;
using System.Collections;

class Test
{
    static void Main() {

        Expression e = new Operation(
            new VariableReference("x"),
            '*',
            new Operation(
                new VariableReference("y"),
                '+',
                new Constant(2)
            )
        );

        Hashtable vars = new Hashtable();

        vars["x"] = 3;
        vars["y"] = 5;
        Console.WriteLine(e.Evaluate(vars));    // Outputs "21"

        vars["x"] = 1.5;
        vars["y"] = 9;
        Console.WriteLine(e.Evaluate(vars));    // Outputs "16.5"
    }
}
```


www.it-ebooks.info

CHRIS SELLS Virtual functions are a major feature of object-oriented programming that differentiate it from other kinds of programming. For example, if you find yourself doing something like this:

```
double GetHourlyRate(Person p) {
  if( p is Student ) { return 1.0; }
  else if( p is Employee ) { return 10.0; }
  return 0.0;
}
```

You should almost always use a virtual method instead:

```
class Person {
         public virtual double GetHourlyRate() {
            return 0.0;
         }
}
class Student {
         public override double GetHourlyRate() {
            return 1.0;
         }
}
class Employee {
         public override double GetHourlyRate() {
            return 10.0;
         }
}
```

1.6.6.5 *Method Overloading*

Method ***overloading*** permits multiple methods in the same class to have the same name as long as they have unique signatures. When compiling an invocation of an overloaded method, the compiler uses ***overload resolution*** to determine the specific method to invoke. Overload resolution finds the one method that best matches the arguments or reports an error if no single best match can be found. The following example shows overload resolution in effect. The comment for each invocation in the `Main` method shows which method is actually invoked.

```
class Test
{
    static void F() {
        Console.WriteLine("F()");
    }

    static void F(object x) {
        Console.WriteLine("F(object)");
    }
```


www.it-ebooks.info

```
    static void F(int x) {
        Console.WriteLine("F(int)");
    }

    static void F(double x) {
        Console.WriteLine("F(double)");
    }

    static void F<T>(T x) {
        Console.WriteLine("F<T>(T)");
    }

    static void F(double x, double y) {
        Console.WriteLine("F(double, double)");
    }

    static void Main() {
        F();                    // Invokes F()
        F(1);                   // Invokes F(int)
        F(1.0);                 // Invokes F(double)
        F("abc");               // Invokes F(object)
        F((double)1);           // Invokes F(double)
        F((object)1);           // Invokes F(object)
        F<int>(1);              // Invokes F<T>(T)
        F(1, 1);                // Invokes F(double, double)
        }
}
```

As shown by the example, a particular method can always be selected by explicitly casting the arguments to the exact parameter types and/or explicitly supplying type arguments.

BRAD ABRAMS The method overloading feature can be abused. Generally speaking, it is better to use method overloading only when all of the methods do semantically the same thing. The way many developers on the consuming end think about method overloading is that a single method takes a variety of arguments. In fact, changing the type of a local variable, parameter, or property could cause a different overload to be called. Developers certainly should not see side effects of the decision to use overloading. For users, however, it can be a surprise when methods with the same name do different things. For example, in the early days of the .NET Framework (before version 1 shipped), we had this set of overloads on the `string` class:

```
public class String {
    public int IndexOf (string value);
         // Returns the index of value with this instance
    public int IndexOf (char value);
         // Returns the index of value with this instance
    public int IndexOf (char [] value);
         // Returns the first index of any of the
         // characters in value within the current instance
}
```

Continued

www.it-ebooks.info

This last overload caused problems, as it does a different thing. For example,

```
"Joshua, Hannah, Joseph".IndexOf("Hannah");// Returns 7
```

but

```
"Joshua, Hannah, Joseph".IndexOf(new char [] {'H','a','n','n','a,'h;");
// Returns 3
```

In this case, it would be better to give the overload that does something a different name:

```
public class String {
    public int IndexOf (string value);
         // Returns the index of value within this instance
    public int IndexOf (char value);
         // Returns the index of value within this instance
    public int IndexOfAny(char [] value);
         // Returns the first index of any of the
         // characters in value within the current instance
}
```

BILL WAGNER Method overloading and inheritance don't mix very well. Because overload resolution rules sometimes favor methods declared in the most derived class, that can sometimes mean a method declared in the derived class may be chosen instead of a method that appears to be a better match in the base class. For that reason, I recommend not overloading members that are declared in a base class.

1.6.7 Other Function Members

Members that contain executable code are collectively known as the ***function members*** of a class. The preceding section describes methods, which are the primary kind of function members. This section describes the other kinds of function members supported by C#: constructors, properties, indexers, events, operators, and destructors.

The following table shows a generic class called `List<T>`, which implements a growable list of objects. The class contains several examples of the most common kinds of function members.

www.it-ebooks.info

```
public class List<T>
{
```

`const int defaultCapacity = 4;`	Constant
`T[] items;` `int count;`	Fields

`public List(int capacity = defaultCapacity) {`
`  items = new T[capacity];`
`}`
	Constructors

`public int Count {`
`  get { return count; }`
`}`
`public int Capacity {`
`  get {`
`  return items.Length;`
`  }`
`  set {`
`  if (value < count) value = count;`
`  if (value != items.Length) {`
`  T[] newItems = new T[value];`
`  Array.Copy(items, 0, newItems, 0, count);`
`  items = newItems;`
`  }`
`  }`
`}`
	Properties

`public T this[int index] {`
`  get {`
`  return items[index];`
`  }`
`  set {`
`  items[index] = value;`
`  OnChanged();`
`  }`
`}`
	Indexer

Continued

www.it-ebooks.info

```
    public void Add(T item) {
        if (count == Capacity) Capacity = count * 2;
        items[count] = item;
        count++;
        OnChanged();
    }
    protected virtual void OnChanged() {
        if (Changed != null) Changed(this, EventArgs.Empty);
    }
    public override bool Equals(object other) {
        return Equals(this, other as List<T>);
    }
    static bool Equals(List<T> a, List<T> b) {
        if (a == null) return b == null;
        if (b == null || a.count != b.count) return false;
        for (int i = 0; i < a.count; i++) {
            if (!object.Equals(a.items[i], b.items[i])) {
                return false;
            }
        }
    return true;
    }
```

Methods

```
    public event EventHandler Changed;
```

Event

```
    public static bool operator ==(List<T> a, List<T> b) {
        return Equals(a, b);
    }
    public static bool operator !=(List<T> a, List<T> b) {
        return !Equals(a, b);
    }
```

Operators

```
}
```

1.6.7.1 *Constructors*

C# supports both instance and static constructors. An ***instance constructor*** is a member that implements the actions required to initialize an instance of a class. A ***static constructor*** is a member that implements the actions required to initialize a class itself when it is first loaded.

A constructor is declared like a method with no return type and the same name as the containing class. If a constructor declaration includes a `static` modifier, it declares a static constructor. Otherwise, it declares an instance constructor.

www.it-ebooks.info

Instance constructors can be overloaded. For example, the `List<T>` class declares two instance constructors, one with no parameters and one that takes an `int` parameter. Instance constructors are invoked using the `new` operator. The following statements allocate two `List<string>` instances using each of the constructors of the `List` class.

```
List<string> list1 = new List<string>();
List<string> list2 = new List<string>(10);
```

Unlike other members, instance constructors are not inherited, and a class has no instance constructors other than those actually declared in the class. If no instance constructor is supplied for a class, then an empty one with no parameters is automatically provided.

BRAD ABRAMS Constructors should be lazy! The best practice is to do minimal work in the constructor—that is, to simply capture the arguments for later use. For example, you might capture the name of the file or the path to the database, but don't open those external resources until absolutely necessary. This practice helps to ensure that possibly scarce resources are allocated for the smallest amount of time possible.

I was personally bitten by this issue recently with the `DataContext` class in Linq to Entities. It opens the database in the connection string provided, rather than waiting to perform that operation until it is needed. For my test cases, I was providing test suspect data directly and, in fact, never wanted to open the database. Not only does this unnecessary activity lead to a performance loss, but it also makes the scenario more complicated.

1.6.7.2 *Properties*

Properties are a natural extension of fields. Both are named members with associated types, and the syntax for accessing fields and properties is the same. However, unlike fields, properties do not denote storage locations. Instead, properties have ***accessors*** that specify the statements to be executed when their values are read or written.

JESSE LIBERTY A property looks to the creator of the class like a method allowing the developer to add behavior prior to setting or retrieving the underlying value. In contrast, the property appears to the client of the class as if it were a field, providing direct, unencumbered access through the assignment operator.

www.it-ebooks.info

ERIC LIPPERT A standard "best practice" is to always expose field-like data as properties with getters and setters rather than exposing the field. That way, if you ever want to add functionality to your getter and setter (e.g., logging, data binding, security checking), you can easily do so without "breaking" any consumer of the code that might rely on the field always being there.

Although in some sense this practice is a violation of another bit of good advice ("Avoid premature generalization"), the new "automatically implemented properties" feature makes it very easy and natural to use properties rather than fields as part of the public interface of a type.

CHRIS SELLS Eric makes such a good point that I wanted to show an example. Don't ever make a field `public`:

```
class Cow
{
     public int Milk; // BAD!
}
```

If you don't want to layer in anything besides storage, let the compiler implement the property for you:

```
class Cow
{
    public int Milk { get; set; } // Good
}
```

That way, the client binds to the property getter and setter so that later you can take over the compiler's implementation to do something fancy:

```
class Cow {
  bool gotMilk = false;
  int milk;
  public int Milk {
   get {
      if( !gotMilk ) {
        milk = ApplyMilkingMachine();
        gotMilk = true; }
      return milk;
    }
    set {
```


www.it-ebooks.info

```
        ApplyReverseMilkingMachine(value); // The cow might not like this..
        milk = value;
      }
    }
    ...
  }
```

Also, I really love the following idiom for cases where you know a calculated value will be used in your program:

```
class Cow {
  public Cow() {
    Milk = ApplyMilkingMachine();
  }


  public int Milk { get; private set; }
  ...
}
```

In this case, we are precalculating the property, which is a waste if we don't know whether we will need it. If we do know, we save ourselves some complication in the code by eliminating a flag, some branching logic, and the storage management.

BILL WAGNER Property accesses look like field accesses to your users—and they will naturally expect them to act like field accesses in every way, including performance. If a `get` accessor needs to do significant work (reading a file or querying a database, for example), it should be exposed as a method, not a property. Callers expect that a method may be doing more work.

For the same reason, repeated calls to property accessors (without intervening code) should return the same value. `DateTime.Now` is one of very few examples in the framework that does not follow this advice.

A property is declared like a field, except that the declaration ends with a `get` accessor and/or a `set` accessor written between the delimiters `{` and `}` instead of ending in a semicolon. A property that has both a `get` accessor and a `set` accessor is a ***read-write property***, a property that has only a `get` accessor is a ***read-only property***, and a property that has only a `set` accessor is a ***write-only property***.

A `get` accessor corresponds to a parameterless method with a return value of the property type. Except as the target of an assignment, when a property is referenced in an expression, the `get` accessor of the property is invoked to compute the value of the property.

www.it-ebooks.info

A `set` accessor corresponds to a method with a single parameter named `value` and no return type. When a property is referenced as the target of an assignment or as the operand of `++` or `--`, the `set` accessor is invoked with an argument that provides the new value.

The `List<T>` class declares two properties, `Count` and `Capacity`, which are read-only and read-write, respectively. The following is an example of use of these properties.

```
List<string> names = new List<string>();
names.Capacity = 100;               // Invokes set accessor
int i = names.Count;                // Invokes get accessor
int j = names.Capacity;             // Invokes get accessor
```

Similar to fields and methods, C# supports both instance properties and static properties. Static properties are declared with the `static` modifier, and instance properties are declared without it.

The accessor(s) of a property can be virtual. When a property declaration includes a `virtual`, `abstract`, or `override` modifier, it applies to the accessor(s) of the property.

> **VLADIMIR RESHETNIKOV** If a virtual property happens to have a `private` accessor, this accessor is implemented in CLR as a nonvirtual method and cannot be overridden in derived classes.

1.6.7.3 *Indexers*

An ***indexer*** is a member that enables objects to be indexed in the same way as an array. An indexer is declared like a property except that the name of the member is `this` followed by a parameter list written between the delimiters `[` and `]`. The parameters are available in the accessor(s) of the indexer. Similar to properties, indexers can be read-write, read-only, and write-only, and the accessor(s) of an indexer can be virtual.

The `List` class declares a single read-write indexer that takes an `int` parameter. The indexer makes it possible to index `List` instances with `int` values. For example:

```
List<string> names = new List<string>();
names.Add("Liz");
names.Add("Martha");
names.Add("Beth");
for (int i = 0; i < names.Count; i++) {
    string s = names[i];
    names[i] = s.ToUpper();
}
```


www.it-ebooks.info

Indexers can be overloaded, meaning that a class can declare multiple indexers as long as the number or types of their parameters differ.

1.6.7.4 *Events*

An ***event*** is a member that enables a class or object to provide notifications. An event is declared like a field except that the declaration includes an `event` keyword and the type must be a delegate type.

JESSE LIBERTY In truth, `event` is just a keyword that signals C# to restrict the way a delegate can be used, thereby preventing a client from directly invoking an event or hijacking an event by assigning a handler rather than adding a handler. In short, the keyword `event` makes delegates behave in the way you expect events to behave.

CHRIS SELLS Without the `event` keyword, you are allowed to do this:

```
delegate void WorkCompleted();

class Worker {
  public WorkCompleted Completed;   // Delegate field, not event
  ...
}

class Boss {
  public void WorkCompleted() { ... }
}

class Program {
  static void Main() {
    Worker peter = new Worker();
    Boss boss = new Boss();

   peter.Completed += boss.WorkCompleted;  // This is what you want to happen
    peter.Completed = boss.WorkCompleted;  // This is what the compiler allows
    ...
  }
}
```

Continued

www.it-ebooks.info

Unfortunately, with the `event` keyword, `Completed` is just a `public` field of type delegate, which can be stepped on by anyone who wants to—and the compiler is okay with that. By adding the `event` keyword, you limit the operations to `+=` and `-=` like so:

```
class Worker {
  public event WorkCompleted Completed;
  ...
}
...
    peter.Completed += boss.WorkCompleted;  // Compiler still okay
    peter.Completed = boss.WorkCompleted;   // Compiler error
```

The use of the `event` keyword is the one time where it's okay to make a field `public`, because the compiler narrows the use to safe operations. Further, if you want to take over the implementation of `+=` and `-+` for an event, you can do so.

Within a class that declares an event member, the event can be accessed like a field of a delegate type (provided the event is not abstract and does not declare accessors). The field stores a reference to a delegate that represents the event handlers that have been added to the event. If no event handlers are present, the field is `null`.

The `List<T>` class declares a single event member called `Changed`, which indicates that a new item has been added to the list. The `Changed` event is raised by the `OnChanged` virtual method, which first checks whether the event is `null` (meaning that no handlers are present). The notion of raising an event is precisely equivalent to invoking the delegate represented by the event—thus there are no special language constructs for raising events.

Clients react to events through ***event handlers***. Event handlers are attached using the `+=` operator and removed using the `-=` operator. The following example attaches an event handler to the `Changed` event of a `List<string>`.

```
using System;

class Test
{
    static int changeCount;

    static void ListChanged(object sender, EventArgs e) {
        changeCount++;
    }
```


www.it-ebooks.info

```
    static void Main() {
        List<string> names = new List<string>();
        names.Changed += new EventHandler(ListChanged);
        names.Add("Liz");
        names.Add("Martha");
        names.Add("Beth");
        Console.WriteLine(changeCount);     // Outputs "3"
    }
}
```

For advanced scenarios where control of the underlying storage of an event is desired, an event declaration can explicitly provide `add` and `remove` accessors, which are somewhat similar to the `set` accessor of a property.

CHRIS SELLS As of C# 2.0, explicitly creating a delegate instance to wrap a method was no longer necessary. As a consequence, the code

```
names.Changed += new EventHandler(ListChanged);
```

can be more succinctly written as

```
names.Changed += ListChanged;
```

Not only does this shortened form require less typing, but it is also easier to read.

1.6.7.5 *Operators*

An ***operator*** is a member that defines the meaning of applying a particular expression operator to instances of a class. Three kinds of operators can be defined: unary operators, binary operators, and conversion operators. All operators must be declared as `public` and `static`.

The `List<T>` class declares two operators, `operator ==` and `operator !=`, and thus gives new meaning to expressions that apply those operators to `List` instances. Specifically, the operators define equality of two `List<T>` instances as comparing each of the contained objects using their `Equals` methods. The following example uses the `==` operator to compare two `List<int>` instances.

```
using System;

class Test
{
    static void Main() {
        List<int> a = new List<int>();
        a.Add(1);
```


www.it-ebooks.info

```
            a.Add(2);
            List<int> b = new List<int>();
            b.Add(1);
            b.Add(2);
            Console.WriteLine(a == b);          // Outputs "True"
            b.Add(3);
            Console.WriteLine(a == b);          // Outputs "False"
        }
    }
```

The first `Console.WriteLine` outputs `True` because the two lists contain the same number of objects with the same values in the same order. Had `List<T>` not defined `operator ==`, the first `Console.WriteLine` would have output `False` because `a` and `b` reference different `List<int>` instances.

1.6.7.6 *Destructors*

A ***destructor*** is a member that implements the actions required to destruct an instance of a class. Destructors cannot have parameters, they cannot have accessibility modifiers, and they cannot be invoked explicitly. The destructor for an instance is invoked automatically during garbage collection.

The garbage collector is allowed wide latitude in deciding when to collect objects and run destructors. Specifically, the timing of destructor invocations is not deterministic, and destructors may be executed on any thread. For these and other reasons, classes should implement destructors only when no other solutions are feasible.

> **VLADIMIR RESHETNIKOV** Destructors are sometimes called "finalizers." This name also appears in the garbage collector API—for example, `GC.WaitForPendingFinalizers`.

The `using` statement provides a better approach to object destruction.

1.7 Structs

Like classes, ***structs*** are data structures that can contain data members and function members, but unlike classes, structs are value types and do not require heap allocation. A variable of a struct type directly stores the data of the struct, whereas a variable of a class type stores a reference to a dynamically allocated object. Struct types do not support user-specified inheritance, and all struct types implicitly inherit from type `object`.

www.it-ebooks.info

ERIC LIPPERT The fact that structs do not *require* heap allocation does *not* mean that they are *never* heap allocated. See the annotations to §1.3 for more details.

Structs are particularly useful for small data structures that have value semantics. Complex numbers, points in a coordinate system, or key–value pairs in a dictionary are all good examples of structs. The use of structs rather than classes for small data structures can make a large difference in the number of memory allocations an application performs. For example, the following program creates and initializes an array of 100 points. With `Point` implemented as a class, 101 separate objects are instantiated—one for the array and one each for the 100 elements.

```
class Point
{
    public int x, y;

    public Point(int x, int y)
    {
        this.x = x;
        this.y = y;
    }
}

class Test
{
    static void Main()
    {
        Point[] points = new Point[100];
        for (int i = 0; i < 100; i++) points[i] = new Point(i, i);
    }
}
```

An alternative is to make `Point` a struct.

```
    struct Point
    {
        public int x, y;
        public Point(int x, int y)

        {
            this.x = x;
            this.y = y;
        }
    }
```

Now, only one object is instantiated—the one for the array—and the `Point` instances are stored in-line in the array.

www.it-ebooks.info

ERIC LIPPERT The takeaway message here is that certain specific data-intensive applications, which would otherwise be gated on heap allocation performance, benefit greatly from using structs. The takeaway message is emphatically *not* "Always use structs because they make your program faster."

The performance benefit here is a tradeoff: Structs can in some scenarios take less time to allocate and deallocate, but because every assignment of a struct is a value copy, they can take more time to copy than a reference copy would take.

Always remember that it makes little sense to optimize anything other than the *slowest* thing. If your program is not gated on heap allocations, then pondering whether to use structs or classes for performance reasons is not an effective use of your time. Find the slowest thing, and then optimize it.

Struct constructors are invoked with the `new` operator, but that does not imply that memory is being allocated. Instead of dynamically allocating an object and returning a reference to it, a struct constructor simply returns the struct value itself (typically in a temporary location on the stack), and this value is then copied as necessary.

With classes, it is possible for two variables to reference the same object and thus possible for operations on one variable to affect the object referenced by the other variable. With structs, the variables each have their own copy of the data, and it is not possible for operations on one to affect the other. For example, the output produced by the following code fragment depends on whether `Point` is a class or a struct.

```
Point a = new Point(10, 10);
Point b = a;
a.x = 20;
Console.WriteLine(b.x);
```

If `Point` is a class, the output is `20` because `a` and `b` reference the same object. If `Point` is a struct, the output is `10` because the assignment of `a` to `b` creates a copy of the value, and this copy is unaffected by the subsequent assignment to `a.x`.

The previous example highlights two of the limitations of structs. First, copying an entire struct is typically less efficient than copying an object reference, so assignment and value parameter passing can be more expensive with structs than with reference types. Second, except for `ref` and `out` parameters, it is not possible to create references to structs, which rules out their usage in a number of situations.

www.it-ebooks.info

BILL WAGNER Read those last two paragraphs again. They describe the most important design differences between structs and classes. If you don't want value semantics in all cases, you must use a class. Classes can implement value semantics in some situations (`string` is a good example), but by default they obey reference semantics. That difference is more important for your designs than size or stack versus heap allocations.

1.8 Arrays

An ***array*** is a data structure that contains a number of variables that are accessed through computed indices. The variables contained in an array, also called the ***elements*** of the array, are all of the same type, and this type is called the ***element type*** of the array.

Array types are reference types, and the declaration of an array variable simply sets aside space for a reference to an array instance. Actual array instances are created dynamically at runtime using the `new` operator. The `new` operation specifies the ***length*** of the new array instance, which is then fixed for the lifetime of the instance. The indices of the elements of an array range from `0` to `Length - 1`. The `new` operator automatically initializes the elements of an array to their default value, which, for example, is zero for all numeric types and `null` for all reference types.

ERIC LIPPERT The confusion resulting from some languages indexing arrays starting with `1` and some others starting with `0` has befuddled multiple generations of novice programmers. The idea that array "indexes" start with `0` comes from a subtle misinterpretation of the C language's array syntax.

In C, when you say `myArray[x]`, what this means is "start at the beginning of the array and refer to the thing `x` steps away." Therefore, `myArray[1]` refers to the *second* element, because that is what you get when you start at the first element and *move* one step.

Really, these references should be called array *offsets* rather than *indices*. But because generations of programmers have now internalized that arrays are "indexed" starting at `0`, we're stuck with this terminology.

www.it-ebooks.info

The following example creates an array of `int` elements, initializes the array, and prints out the contents of the array.

```
using System;

class Test
{
    static void Main() {
        int[] a = new int[10];
        for (int i = 0; i < a.Length; i++) {
            a[i] = i * i;
        }
        for (int i = 0; i < a.Length; i++) {
            Console.WriteLine("a[{0}] = {1}", i, a[i]);
        }
    }
}
```

This example creates and operates on a ***single-dimensional array***. C# also supports ***multi-dimensional arrays***. The number of dimensions of an array type, also known as the ***rank*** of the array type, is one plus the number of commas written between the square brackets of the array type. The following example allocates one-dimensional, two-dimensional, and three-dimensional arrays.

```
int[] a1 = new int[10];
int[,] a2 = new int[10, 5];
int[,,] a3 = new int[10, 5, 2];
```

The `a1` array contains 10 elements, the `a2` array contains 50 (10 × 5) elements, and the `a3` array contains 100 (10 × 5 × 2) elements.

> **BILL WAGNER** An FxCop rule recommends against multi-dimensional arrays; it's primarily guidance against using multi-dimensional arrays as sparse arrays. If you know that you really are filling in all the elements in the array, multi-dimensional arrays are fine.

The element type of an array can be any type, including an array type. An array with elements of an array type is sometimes called a ***jagged array*** because the lengths of the element arrays do not all have to be the same. The following example allocates an array of arrays of `int`:

```
int[][] a = new int[3][];
a[0] = new int[10];
a[1] = new int[5];
a[2] = new int[20];
```

The first line creates an array with three elements, each of type `int[]` and each with an initial value of `null`. The subsequent lines then initialize the three elements with references to individual array instances of varying lengths.

www.it-ebooks.info

The new operator permits the initial values of the array elements to be specified using an *array initializer*, which is a list of expressions written between the delimiters { and }. The following example allocates and initializes an int[] with three elements.

```
int[] a = new int[] {1, 2, 3};
```

Note that the length of the array is inferred from the number of expressions between { and }. Local variable and field declarations can be shortened further such that the array type does not have to be restated.

```
int[] a = {1, 2, 3};
```

Both of the previous examples are equivalent to the following:

```
int[] t = new int[3];
t[0] = 1;
t[1] = 2;
t[2] = 3;
int[] a = t;
```

ERIC LIPPERT In a number of places thus far, the specification notes that a particular local initialization is equivalent to "assign something to a temporary variable, do something to the temporary variable, declare a local variable, and assign the temporary to the local variable." You may be wondering why the specification calls out this seemingly unnecessary indirection. Why not simply say that this initialization is equivalent to this:

```
int[] a = new int[3];
a[0] = 1; a[1] = 2; a[2] =3;
```

In fact, this practice is necessary because of definite assignment analysis. We would like to ensure that all local variables are definitely assigned before they are used. In particular, we would like an expression such as `object[] arr = {arr};` to be illegal because it appears to use arr before it is definitely assigned. If this were equivalent to

```
object[] arr = new object[1];
arr[0] = arr;
```

then that would be legal. But by saying that this expression is equivalent to

```
object[] temp = new object[1];
temp[0] = arr;
object[] arr = temp;
```

then it becomes clear that arr is being used before it is assigned.

www.it-ebooks.info

1.9 Interfaces

An ***interface*** defines a contract that can be implemented by classes and structs. An interface can contain methods, properties, events, and indexers. An interface does not provide implementations of the members it defines—it merely specifies the members that must be supplied by classes or structs that implement the interface.

Interfaces may employ ***multiple inheritance***. In the following example, the interface `IComboBox` inherits from both `ITextBox` and `IListBox`.

```
interface IControl
{
    void Paint();
}

interface ITextBox : IControl
{
    void SetText(string text);
}

interface IListBox : IControl
{
    void SetItems(string[] items);
}

interface IComboBox : ITextBox, IListBox { }
```

Classes and structs can implement multiple interfaces. In the following example, the class `EditBox` implements both `IControl` and `IDataBound`.

```
interface IDataBound
{
    void Bind(Binder b);
}

public class EditBox : IControl, IDataBound
{
    public void Paint() {...}

    public void Bind(Binder b) {...}
}
```

KRZYSZTOF CWALINA Perhaps I am stirring up quite a bit of controversy with this statement, but I believe the lack of support for multiple inheritance in our type system is the single biggest contributor to the complexity of the .NET Framework. When we designed the type system, we explicitly decided not to add support for multiple inheritance so as to provide simplicity. In retrospect, this decision had the exact opposite effect. The lack of multiple inheritance forced us to add the concept of interfaces, which in turn are responsible for problems with the evolution of the framework, deeper inheritance hierarchies, and many other problems.

www.it-ebooks.info

When a class or struct implements a particular interface, instances of that class or struct can be implicitly converted to that interface type. For example:

```
EditBox editBox = new EditBox();
IControl control = editBox;
IDataBound dataBound = editBox;
```

In cases where an instance is not statically known to implement a particular interface, dynamic type casts can be used. For example, the following statements use dynamic type casts to obtain an object's `IControl` and `IDataBound` interface implementations. Because the actual type of the object is `EditBox`, the casts succeed.

```
object obj = new EditBox();
IControl control = (IControl)obj;
IDataBound dataBound = (IDataBound)obj;
```

In the previous `EditBox` class, the `Paint` method from the `IControl` interface and the `Bind` method from the `IDataBound` interface are implemented using `public` members. C# also supports ***explicit interface member implementations***, using which the class or struct can avoid making the members `public`. An explicit interface member implementation is written using the fully qualified interface member name. For example, the `EditBox` class could implement the `IControl.Paint` and `IDataBound.Bind` methods using explicit interface member implementations as follows.

```
public class EditBox : IControl, IDataBound
{
    void IControl.Paint() {...}

    void IDataBound.Bind(Binder b) {...}
}
```

Explicit interface members can only be accessed via the interface type. For example, the implementation of `IControl.Paint` provided by the previous `EditBox` class can only be invoked by first converting the `EditBox` reference to the `IControl` interface type.

```
EditBox editBox = new EditBox();
editBox.Paint();                        // Error; no such method
IControl control = editBox;
control.Paint();                        // Okay
```

VLADIMIR RESHETNIKOV Actually, explicitly implemented interface members can also be accessed via a type parameter, constrained to the interface type.

www.it-ebooks.info

1.10 Enums

An ***enum type*** is a distinct value type with a set of named constants. The following example declares and uses an enum type named `Color` with three constant values, `Red`, `Green`, and `Blue`.

```
using System;

enum Color
{
   Red,
   Green,
   Blue
}

class Test
{
    static void PrintColor(Color color) {
        switch (color) {
            case Color.Red:
                Console.WriteLine("Red");
                break;
            case Color.Green:
                Console.WriteLine("Green");
                break;
            case Color.Blue:
                Console.WriteLine("Blue");
                break;
            default:
                Console.WriteLine("Unknown color");
                break;
        }
    }

    static void Main() {
        Color c = Color.Red;
        PrintColor(c);
        PrintColor(Color.Blue);
    }
}
```

Each enum type has a corresponding integral type called the ***underlying type*** of the enum type. An enum type that does not explicitly declare an underlying type has an underlying type of `int`. An enum type's storage format and range of possible values are determined by its underlying type. The set of values that an enum type can take on is not limited by its enum members. In particular, any value of the underlying type of an enum can be cast to the enum type and is a distinct valid value of that enum type.

The following example declares an enum type named `Alignment` with an underlying type of `sbyte`.

```
enum Alignment : sbyte
{
    Left = -1,
```


www.it-ebooks.info

```
        Center = 0,
        Right = 1
    }
```

VLADIMIR RESHETNIKOV Although this syntax resembles base type specification, it has a different meaning. The base type of `Alignment` is not `sbyte`, but `System.Enum`, and there is no implicit conversion from `Alignment` to `sbyte`.

As shown by the previous example, an enum member declaration can include a constant expression that specifies the value of the member. The constant value for each enum member must be in the range of the underlying type of the enum. When an enum member declaration does not explicitly specify a value, the member is given the value zero (if it is the first member in the enum type) or the value of the textually preceding enum member plus one.

Enum values can be converted to integral values and vice versa using type casts. For example:

```
int i = (int)Color.Blue;          // int i = 2;
Color c = (Color)2;               // Color c = Color.Blue;
```

BILL WAGNER The fact that zero is the default value for a variable of an enum type implies that you should always ensure that zero is a valid member of any enum type.

The default value of any enum type is the integral value zero converted to the enum type. In cases where variables are automatically initialized to a default value, this is the value given to variables of enum types. For the default value of an enum type to be easily available, the literal `0` implicitly converts to any enum type. Thus the following is permitted.

```
Color c = 0;
```

BRAD ABRAMS My first programming class in high school was in Turbo Pascal (Thanks, Anders!). On one of my first assignments I got back from my teacher, I saw a big red circle around the number 65 in my source code and the scrawled note, "No Magic Constants!!" My teacher was instilling in me the virtues of using the constant `RetirementAge` for readability and maintenance. Enums make this a super-easy decision to make. Unlike in some programming languages, using an enum does not incur any runtime performance overhead in C#. While I have heard many excuses in API reviews, there are just no good reasons to use a magic constant rather than an enum!

www.it-ebooks.info

1.11 Delegates

A ***delegate type*** represents references to methods with a particular parameter list and return type. Delegates make it possible to treat methods as entities that can be assigned to variables and passed as parameters. Delegates are similar to the concept of function pointers found in some other languages, but unlike function pointers, delegates are object-oriented and type-safe.

The following example declares and uses a delegate type named `Function`.

```
using System;

delegate double Function(double x);

class Multiplier
{
    double factor;

    public Multiplier(double factor) {
        this.factor = factor;
    }

    public double Multiply(double x) {
        return x * factor;
    }
}

class Test
{
    static double Square(double x) {
        return x * x;
    }

    static double[] Apply(double[] a, Function f) {
        double[] result = new double[a.Length];
        for (int i = 0; i < a.Length; i++) result[i] = f(a[i]);
        return result;
    }

    static void Main() {
        double[] a = {0.0, 0.5, 1.0};

        double[] squares = Apply(a, Square);

        double[] sines = Apply(a, Math.Sin);

        Multiplier m = new Multiplier(2.0);
        double[] doubles =  Apply(a, m.Multiply);
    }
}
```

An instance of the `Function` delegate type can reference any method that takes a `double` argument and returns a `double` value. The `Apply` method applies a given `Function` to the elements of a `double[]`, returning a `double[]` with the results. In the `Main` method, `Apply` is used to apply three different functions to a `double[]`.

www.it-ebooks.info

A delegate can reference either a static method (such as `Square` or `Math.Sin` in the previous example) or an instance method (such as `m.Multiply` in the previous example). A delegate that references an instance method also references a particular object, and when the instance method is invoked through the delegate, that object becomes `this` in the invocation.

Delegates can also be created using anonymous functions, which are "in-line methods" that are created on the fly. Anonymous functions can see the local variables of the surrounding methods. Thus the multiplier example above can be written more easily without using a `Multiplier` class:

```
double[] doubles =  Apply(a, (double x) => x * 2.0);
```

An interesting and useful property of a delegate is that it does not know or care about the class of the method it references; all that matters is that the referenced method has the same parameters and return type as the delegate.

BILL WAGNER This property of delegates make them an excellent tool for providing interfaces between components with the lowest possible coupling.

1.12 Attributes

Types, members, and other entities in a C# program support modifiers that control certain aspects of their behavior. For example, the accessibility of a method is controlled using the `public`, `protected`, `internal`, and `private` modifiers. C# generalizes this capability such that user-defined types of declarative information can be attached to program entities and retrieved at runtime. Programs specify this additional declarative information by defining and using ***attributes***.

The following example declares a `HelpAttribute` attribute that can be placed on program entities to provide links to their associated documentation.

```
using System;

public class HelpAttribute: Attribute
{
    string url;
    string topic;

    public HelpAttribute(string url) {
        this.url = url;
    }

    public string Url {
        get { return url; }
    }
```


www.it-ebooks.info

```
    public string Topic {
        get { return topic; }
        set { topic = value; }
    }
}
```

All attribute classes derive from the `System.Attribute` base class provided by the .NET Framework. Attributes can be applied by giving their name, along with any arguments, inside square brackets just before the associated declaration. If an attribute's name ends in `Attribute`, that part of the name can be omitted when the attribute is referenced. For example, the `HelpAttribute` attribute can be used as follows.

```
[Help("http://msdn.microsoft.com/.../MyClass.htm")]
public class Widget
{
   [Help("http://msdn.microsoft.com/.../MyClass.htm", Topic = "Display")]
    public void Display(string text) { }
}
```

This example attaches a `HelpAttribute` to the `Widget` class and another `HelpAttribute` to the `Display` method in the class. The public constructors of an attribute class control the information that must be provided when the attribute is attached to a program entity. Additional information can be provided by referencing public read-write properties of the attribute class (such as the reference to the `Topic` property previously).

The following example shows how attribute information for a given program entity can be retrieved at runtime using reflection.

```
using System;
using System.Reflection;

class Test
{
    static void ShowHelp(MemberInfo member) {
        HelpAttribute a = Attribute.GetCustomAttribute(member,
            typeof(HelpAttribute)) as HelpAttribute;
        if (a == null) {
            Console.WriteLine("No help for {0}", member);
        }
        else {
            Console.WriteLine("Help for {0}:", member);
            Console.WriteLine("  Url={0}, Topic={1}",
                              a.Url, a.Topic);
        }
    }

    static void Main() {
        ShowHelp(typeof(Widget));
        ShowHelp(typeof(Widget).GetMethod("Display"));
    }
}
```


www.it-ebooks.info

When a particular attribute is requested through reflection, the constructor for the attribute class is invoked with the information provided in the program source, and the resulting attribute instance is returned. If additional information was provided through properties, those properties are set to the given values before the attribute instance is returned.

BILL WAGNER The full potential of attributes will be realized when some future version of the C# compiler enables developers to read attributes and use them to modify the code model before the compiler creates IL. I've wanted to be able to use attributes to change the behavior of code since the first release of C#.

www.it-ebooks.info

This page intentionally left blank

www.it-ebooks.info

2. Lexical Structure

2.1 Programs

A C# ***program*** consists of one or more ***source files***, known formally as ***compilation units*** (§9.1). A source file is an ordered sequence of Unicode characters. Source files typically have a one-to-one correspondence with files in a file system, but this correspondence is not required. For maximal portability, it is recommended that files in a file system be encoded with the UTF-8 encoding.

Conceptually speaking, a program is compiled using three steps:

1. Transformation, which converts a file from a particular character repertoire and encoding scheme into a sequence of Unicode characters.
2. Lexical analysis, which translates a stream of Unicode input characters into a stream of tokens.
3. Syntactic analysis, which translates the stream of tokens into executable code.

2.2 Grammars

This specification presents the syntax of the C# programming language using two grammars. The ***lexical grammar*** (§2.2.2) defines how Unicode characters are combined to form line terminators, white space, comments, tokens, and preprocessing directives. The ***syntactic grammar*** (§2.2.3) defines how the tokens resulting from the lexical grammar are combined to form C# programs.

2.2.1 Grammar Notation

The lexical and syntactic grammars are presented using ***grammar productions***. Each grammar production defines a nonterminal symbol and the possible expansions of that nonterminal symbol into sequences of nonterminal or terminal symbols. In grammar productions, *nonterminal* symbols are shown in italic type, and `terminal` symbols are shown in a fixed-width font.

www.it-ebooks.info

The first line of a grammar production is the name of the nonterminal symbol being defined, followed by a colon. Each successive indented line contains a possible expansion of the nonterminal given as a sequence of nonterminal or terminal symbols. For example, the production

while-statement:
`while` `(` *boolean-expression* `)` *embedded-statement*

defines a *while-statement* to consist of the token `while`, followed by the token "`(`", followed by a *boolean-expression*, followed by the token "`)`", followed by an *embedded-statement*.

When there is more than one possible expansion of a nonterminal symbol, the alternatives are listed on separate lines. For example, the production

statement-list:
statement
statement-list *statement*

defines a *statement-list* to either consist of a *statement* or consist of a *statement-list* followed by a *statement*. In other words, the definition is recursive and specifies that a statement list consists of one or more statements.

A subscripted suffix "$_{opt}$" is used to indicate an optional symbol. The production

block:
`{` *statement-list*$_{opt}$ `}`

is shorthand for

block:
`{` `}`
`{` *statement-list* `}`

and defines a *block* to consist of an optional *statement-list* enclosed in "`{`" and "`}`" tokens.

Alternatives are normally listed on separate lines, though in cases where there are many alternatives, the phrase "one of" may precede a list of expansions given on a single line. This is simply shorthand for listing each of the alternatives on a separate line. For example, the production

real-type-suffix: one of
`F` `f` `D` `d` `M` `m`

is shorthand for

www.it-ebooks.info

real-type-suffix:
F
f
D
d
M
m

2.2.2 Lexical Grammar

The lexical grammar of C# is presented in §2.3, §2.4, and §2.5. The terminal symbols of the lexical grammar are the characters of the Unicode character set, and the lexical grammar specifies how characters are combined to form tokens (§2.4), white space (§2.3.3), comments (§2.3.2), and preprocessing directives (§2.5).

Every source file in a C# program must conform to the *input* production of the lexical grammar (§2.3).

2.2.3 Syntactic Grammar

The syntactic grammar of C# is presented in the chapters and appendices that follow this chapter. The terminal symbols of the syntactic grammar are the tokens defined by the lexical grammar, and the syntactic grammar specifies how tokens are combined to form C# programs.

Every source file in a C# program must conform to the *compilation-unit* production of the syntactic grammar (§9.1).

2.3 Lexical Analysis

The *input* production defines the lexical structure of a C# source file. Each source file in a C# program must conform to this lexical grammar production.

input:
input-section$_{opt}$

input-section:
input-section-part
input-section *input-section-part*

input-section-part:
input-elements$_{opt}$ *new-line*
pp-directive

www.it-ebooks.info

input-elements:
input-element
input-elements input-element

input-element:
whitespace
comment
token

Five basic elements make up the lexical structure of a C# source file: line terminators (§2.3.1), white space (§2.3.3), comments (§2.3.2), tokens (§2.4), and preprocessing directives (§2.5). Of these basic elements, only tokens are significant in the syntactic grammar of a C# program (§2.2.3).

The lexical processing of a C# source file consists of reducing the file into a sequence of tokens, which then becomes the input to the syntactic analysis. Line terminators, white space, and comments can serve to separate tokens, and preprocessing directives can cause sections of the source file to be skipped, but otherwise these lexical elements have no impact on the syntactic structure of a C# program.

When several lexical grammar productions match a sequence of characters in a source file, the lexical processing always forms the longest possible lexical element. For example, the character sequence `//` is processed as the beginning of a single-line comment because that lexical element is longer than a single `/` token.

2.3.1 Line Terminators

Line terminators divide the characters of a C# source file into lines.

new-line:
Carriage return character (`U+000D`)
Line feed character (`U+000A`)
Carriage return character (`U+000D`) followed by line feed character (`U+000A`)
Next line character (`U+0085`)
Line separator character (`U+2028`)
Paragraph separator character (`U+2029`)

For compatibility with source code editing tools that add end-of-file markers, and to enable a source file to be viewed as a sequence of properly terminated lines, the following transformations are applied, in order, to every source file in a C# program:

- If the last character of the source file is a Control-Z character (`U+001A`), this character is deleted.

www.it-ebooks.info

- A carriage return character (U+000D) is added to the end of the source file if that source file is non-empty and if the last character of the source file is not a carriage return (U+000D), a line feed (U+000A), a line separator (U+2028), or a paragraph separator (U+2029).

2.3.2 Comments

Two forms of comments are supported: single-line comments and delimited comments. ***Single-line comments*** start with the characters // and extend to the end of the source line. ***Delimited comments*** start with the characters /* and end with the characters */. Delimited comments may span multiple lines.

comment:
    single-line-comment
    delimited-comment

single-line-comment:
    // *input-characters*$_{opt}$

input-characters:
    input-character
    input-characters input-character

input-character:
    Any Unicode character except a *new-line-character*

new-line-character:
    Carriage return character (U+000D)
    Line feed character (U+000A)
    Next line character (U+0085)
    Line separator character (U+2028)
    Paragraph separator character (U+2029)

delimited-comment:
    /* *delimited-comment-text*$_{opt}$ *asterisks* /

delimited-comment-text:
    delimited-comment-section
    delimited-comment-text delimited-comment-section

delimited-comment-section:
    /
    asterisks$_{opt}$ *not-slash-or-asterisk*

www.it-ebooks.info

asterisks:
    *
    asterisks *

not-slash-or-asterisk:
    Any Unicode character except / or *

Comments do not nest. The character sequences /* and */ have no special meaning within a // comment, and the character sequences // and /* have no special meaning within a delimited comment.

Comments are not processed within character and string literals.

The example

```
/* Hello, world program
   This program writes "hello, world" to the console
*/
class Hello
{
    static void Main() {
        System.Console.WriteLine("hello, world");
    }
}
```

includes a delimited comment.

The example

```
// Hello, world program
// This program writes "hello, world" to the console
//
class Hello // any name will do for this class
{
    static void Main() { // this method must be named "Main"
        System.Console.WriteLine("hello, world");
    }
}
```

shows several single-line comments.

2.3.3 White Space

White space is defined as any character with Unicode class Zs (which includes the space character) as well as the horizontal tab character, the vertical tab character, and the form feed character.

www.it-ebooks.info

whitespace:
Any character with Unicode class Zs
Horizontal tab character (`U+0009`)
Vertical tab character (`U+000B`)
Form feed character (`U+000C`)

2.4 Tokens

There are several kinds of tokens: identifiers, keywords, literals, operators, and punctuators. White space and comments are not tokens, though they act as separators for tokens.

token:
identifier
keyword
integer-literal
real-literal
character-literal
string-literal
operator-or-punctuator

2.4.1 Unicode Character Escape Sequences

A Unicode character escape sequence represents a Unicode character. Unicode character escape sequences are processed in identifiers (§2.4.2), character literals (§2.4.4.4), and regular string literals (§2.4.4.5). A Unicode character escape is not processed in any other location (for example, to form an operator, punctuator, or keyword).

unicode-escape-sequence:
`\u` *hex-digit hex-digit hex-digit hex-digit*
`\U` *hex-digit hex-digit hex-digit hex-digit hex-digit hex-digit hex-digit hex-digit*

ERIC LIPPERT This practice differs from Java, in which Unicode escape sequences may appear almost anywhere.

A Unicode escape sequence represents the single Unicode character formed by the hexadecimal number following the "`\u`" or "`\U`" characters. Since C# uses a 16-bit encoding of Unicode code points in characters and string values, a Unicode character in the range U+10000 to U+10FFFF is not permitted in a character literal and is represented using a

www.it-ebooks.info

Unicode surrogate pair in a string literal. Unicode characters with code points above 0x10FFFF are not supported.

Multiple translations are not performed. For instance, the string literal "`\u005Cu005C`" is equivalent to "`\u005C`" rather than "\". The Unicode value `\u005C` is the character "\".

The example

```
class Class1
{
    static void Test(bool \u0066) {
        char c = '\u0066';
        if (\u0066)
            System.Console.WriteLine(c.ToString());
    }
}
```

shows several uses of `\u0066`, which is the escape sequence for the letter "`f`". The program is equivalent to

```
class Class1
{
    static void Test(bool f) {
        char c = 'f';
        if (f)
            System.Console.WriteLine(c.ToString());
    }
}
```


2.4.2 Identifiers

The rules for identifiers given in this section correspond exactly to those recommended by the Unicode Standard Annex 31, except that underscore is allowed as an initial character (as is traditional in the C programming language), Unicode escape sequences are permitted in identifiers, and the "`@`" character is allowed as a prefix to enable keywords to be used as identifiers.

identifier:
    available-identifier
    `@` *identifier-or-keyword*

available-identifier:
    An *identifier-or-keyword* that is not a *keyword*

identifier-or-keyword:
    identifier-start-character *identifier-part-characters*$_{opt}$

www.it-ebooks.info

identifier-start-character:
    letter-character
    _ (the underscore character `U+005F`)

identifier-part-characters:
    identifier-part-character
    identifier-part-characters *identifier-part-character*

identifier-part-character:
    letter-character
    decimal-digit-character
    connecting-character
    combining-character
    formatting-character

letter-character:
    A Unicode character of classes Lu, Ll, Lt, Lm, Lo, or Nl
    A *unicode-escape-sequence* representing a character of classes Lu, Ll, Lt, Lm, Lo, or Nl

combining-character:
    A Unicode character of classes Mn or Mc
    A *unicode-escape-sequence* representing a character of classes Mn or Mc

decimal-digit-character:
    A Unicode character of the class Nd
    A *unicode-escape-sequence* representing a character of the class Nd

connecting-character:
    A Unicode character of the class Pc
    A *unicode-escape-sequence* representing a character of the class Pc

formatting-character:
    A Unicode character of the class Cf
    A *unicode-escape-sequence* representing a character of the class Cf

For information on the Unicode character classes mentioned above, see *The Unicode Standard*, *Version 3.0*, section 4.5.

Examples of valid identifiers include "`identifier1`", "`_identifier2`", and "`@if`".

An identifier in a conforming program must be in the canonical format defined by Unicode Normalization Form C, as defined by Unicode Standard Annex 15. The behavior when encountering an identifier not in Normalization Form C is implementation-defined; however, a diagnostic is not required.

www.it-ebooks.info

The prefix "@" enables the use of keywords as identifiers, which is useful when interfacing with other programming languages. The character @ is not actually part of the identifier, so the identifier might be seen in other languages as a normal identifier, without the prefix. An identifier with an @ prefix is called a ***verbatim identifier***. Use of the @ prefix for identifiers that are not keywords is permitted, but strongly discouraged as a matter of style.

The example

```
class @class
{
    public static void @static(bool @bool) {
        if (@bool)
            System.Console.WriteLine("true");
        else
            System.Console.WriteLine("false");
    }
}

class Class1
{
    static void M() {
        cl\u0061ss.st\u0061tic(true);
    }
}
```

defines a class named "`class`" with a static method named "`static`" that takes a parameter named "`bool`". Note that since Unicode escapes are not permitted in keywords, the token "`cl\u0061ss`" is an identifier, and is the same identifier as "`@class`".

Two identifiers are considered the same if they are identical after the following transformations are applied, in order:

- The prefix "@", if used, is removed.
- Each *unicode-escape-sequence* is transformed into its corresponding Unicode character.
- Any *formatting-character*s are removed.

Identifiers containing two consecutive underscore characters (`U+005F`) are reserved for use by the implementation. For example, an implementation might provide extended keywords that begin with two underscores.

2.4.3 Keywords

A ***keyword*** is an identifier-like sequence of characters that is reserved, and cannot be used as an identifier except when prefaced by the @ character.

www.it-ebooks.info

keyword: one of

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

In some places in the grammar, specific identifiers have special meaning, but are not keywords. Such identifiers are sometimes referred to as "contextual keywords." For example, within a property declaration, the "`get`" and "`set`" identifiers have special meaning (§10.7.2). An identifier other than `get` or `set` is never permitted in these locations, so this use does not conflict with a use of these words as identifiers. In other cases, such as with the identifier "`var`" in implicitly typed local variable declarations (§8.5.1), a contextual keyword can conflict with declared names. In such cases, the declared name takes precedence over the use of the identifier as a contextual keyword.

ERIC LIPPERT C# has not added any new reserved keywords since its original release. All the new language features that require new keywords (`yield`, `select`, and so on) use "contextual keywords," which are not reserved and have special meaning only in context. This decision helps preserve backward compatibility with existing programs.

VLADIMIR RESHETNIKOV In case you need the full list of these contextual keywords (including supported attribute targets), here it is:

```
add alias ascending assembly by descending dynamic equals field from get
global group into join let method module on orderby param partial property
remove select set type typevar value var where yield
```


www.it-ebooks.info

2.4.4 Literals

A ***literal*** is a source code representation of a value.

literal:
    boolean-literal
    integer-literal
    real-literal
    character-literal
    string-literal
    null-literal

2.4.4.1 *Boolean Literals*

There are two boolean literal values: `true` and `false`.

boolean-literal:
    `true`
    `false`

The type of a *boolean-literal* is `bool`.

2.4.4.2 *Integer Literals*

Integer literals are used to write values of types `int`, `uint`, `long`, and `ulong`. Integer literals have two possible forms: decimal and hexadecimal.

integer-literal:
    decimal-integer-literal
    hexadecimal-integer-literal

decimal-integer-literal:
    decimal-digits *integer-type-suffix*$_{opt}$

decimal-digits:
    decimal-digit
    decimal-digits *decimal-digit*

decimal-digit: one of
    `0` `1` `2` `3` `4` `5` `6` `7` `8` `9`

integer-type-suffix: one of
    `U` `u` `L` `l` `UL` `Ul` `uL` `ul` `LU` `Lu` `lU` `lu`

www.it-ebooks.info

hexadecimal-integer-literal:
    `0x` *hex-digits* *integer-type-suffix*$_{opt}$
    `0X` *hex-digits* *integer-type-suffix*$_{opt}$

hex-digits:
    hex-digit
    hex-digits *hex-digit*

hex-digit: one of
    `0 1 2 3 4 5 6 7 8 9 A B C D E F a b c d e f`

ERIC LIPPERT C# does not support octal literals, for two reasons. First, hardly anyone uses octal literals these days. Second, if C# supported octal in the standard "leading zero means octal" format, then it would be a potential source of errors. Consider this code:

```
FlightNumber = 0541;
```

Clearly this expression is intended as a decimal literal, not an octal literal.

The type of an integer literal is determined as follows:

- If the literal has no suffix, it has the first of these types in which its value can be represented: `int`, `uint`, `long`, `ulong`.
- If the literal is suffixed by `U` or `u`, it has the first of these types in which its value can be represented: `uint`, `ulong`.
- If the literal is suffixed by `L` or `l`, it has the first of these types in which its value can be represented: `long`, `ulong`.
- If the literal is suffixed by `UL`, `Ul`, `uL`, `ul`, `LU`, `Lu`, `lU`, or `lu`, it is of type `ulong`.

If the value represented by an integer literal is outside the range of the `ulong` type, a compile-time error occurs.

As a matter of style, it is suggested that "`L`" be used instead of "`l`" when writing literals of type `long`, since it is easy to confuse the letter "`l`" with the digit "`1`".

To permit the smallest possible `int` and `long` values to be written as decimal integer literals, the following two rules exist:

- When a *decimal-integer-literal* with the value 2147483648 (2^{31}) and no *integer-type-suffix* appears as the token immediately following a unary minus operator token (§7.7.2), the

www.it-ebooks.info

result is a constant of type `int` with the value –2147483648 (–2^{31}). In all other situations, such a *decimal-integer-literal* is of type `uint`.

- When a *decimal-integer-literal* with the value 9223372036854775808 (2^{63}) and no *integer-type-suffix* or the *integer-type-suffix* `L` or `l` appears as the token immediately following a unary minus operator token (§7.7.2), the result is a constant of type `long` with the value –9223372036854775808 (–2^{63}). In all other situations, such a *decimal-integer-literal* is of type `ulong`.

> **JOSEPH ALBAHARI** Thanks to implicit constant expression conversions (§6.1.8), integer literals can be assigned directly to the `uint`, `long`, and `ulong` types (as well as `short`, `ushort`, `byte`, and `sbyte`):
>
> ```
> uint x = 3; long y = 3; ulong z = 3;
> ```
>
> As a consequence, the `U` and `L` suffixes are rarely necessary. An example of when they are still useful is to force 64-bit calculations on literals that would otherwise attract 32-bit arithmetic:
>
> ```
> long error    = 1000000  * 1000000 ; // Compile-time error (32-bit overflow)
> long trillion = 1000000L * 1000000L; // Okay -- no overflow
> ```


2.4.4.3 *Real Literals*

Real literals are used to write values of types `float`, `double`, and `decimal`.

real-literal:
decimal-digits `.` *decimal-digits* *exponent-part*$_{opt}$ *real-type-suffix*$_{opt}$
`.` *decimal-digits* *exponent-part*$_{opt}$ *real-type-suffix*$_{opt}$
decimal-digits *exponent-part* *real-type-suffix*$_{opt}$
decimal-digits *real-type-suffix*

exponent-part:
`e` *sign*$_{opt}$ *decimal-digits*
`E` *sign*$_{opt}$ *decimal-digits*

sign: one of
`+` `-`

real-type-suffix: one of
`F` `f` `D` `d` `M` `m`

www.it-ebooks.info

If no *real-type-suffix* is specified, the type of the real literal is `double`. Otherwise, the real type suffix determines the type of the real literal, as follows:

- A real literal suffixed by `F` or `f` is of type `float`. For example, the literals `1f`, `1.5f`, `1e10f`, and `123.456F` are all of type `float`.
- A real literal suffixed by `D` or `d` is of type `double`. For example, the literals `1d`, `1.5d`, `1e10d`, and `123.456D` are all of type `double`.
- A real literal suffixed by `M` or `m` is of type `decimal`. For example, the literals `1m`, `1.5m`, `1e10m`, and `123.456M` are all of type `decimal`. This literal is converted to a `decimal` value by taking the exact value and, if necessary, rounding to the nearest representable value using banker's rounding (§4.1.7). Any scale apparent in the literal is preserved unless the value is rounded or the value is zero (in which latter case, the sign and scale will be 0). Hence the literal `2.900m` will be parsed to form the decimal with sign `0`, coefficient `2900`, and scale `3`.

If the specified literal cannot be represented in the indicated type, a compile-time error occurs.

The value of a real literal of type `float` or `double` is determined by using the IEEE "round to nearest" mode.

Note that in a real literal, decimal digits are always required after the decimal point. For example, `1.3F` is a real literal but `1.F` is not.

JOSEPH ALBAHARI Of all the numeric suffixes, `m` and `f` are by far the most useful. Without these suffixes, a fractional `float` or `decimal` literal cannot be specified without a cast. For example, the following code will not compile, because the literal `1.5` will be parsed as type `double`:

```
float x = 1.5;    // Error: no implicit conversion from double to float
decimal y = 1.5;  // Error: no implicit conversion from double to decimal
```

Interestingly, the following code *does* compile, because C# defines an implicit conversion from `int` to `decimal`:

```
decimal z = 123;  // Okay: parsed as int, and then implicitly converted
                  // to decimal
```

The `d` suffix is technically redundant in that the presence of a decimal point does the same job:

```
Console.WriteLine ((123.0).GetType() == typeof (double));// True
```


www.it-ebooks.info

2.4.4.4 *Character Literals*

A character literal represents a single character, and usually consists of a character in quotes, as in `'a'`.

character-literal:
    `'` *character* `'`

character:
    single-character
    simple-escape-sequence
    hexadecimal-escape-sequence
    unicode-escape-sequence

single-character:
    Any character except `'` (`U+0027`), `\` (`U+005C`), and *new-line-character*

simple-escape-sequence: one of
    `\'` `\"` `\\` `\0` `\a` `\b` `\f` `\n` `\r` `\t` `\v`

hexadecimal-escape-sequence:
    `\x` *hex-digit* *hex-digit*$_{opt}$ *hex-digit*$_{opt}$ *hex-digit*$_{opt}$

A character that follows a backslash character (`\`) in a *character* must be one of the following characters: `'`, `"`, `\`, `0`, `a`, `b`, `f`, `n`, `r`, `t`, `u`, `U`, `x`, `v`. Otherwise, a compile-time error occurs.

A hexadecimal escape sequence represents a single Unicode character, with the value formed by the hexadecimal number following "`\x`".

If the value represented by a character literal is greater than `U+FFFF`, a compile-time error occurs.

A Unicode character escape sequence (§2.4.1) in a character literal must be in the range `U+0000` to `U+FFFF`.

A simple escape sequence represents a Unicode character encoding, as described in the table below.

Escape Sequence	Character Name	Unicode Encoding
`\'`	Single quote	`0x0027`
`\"`	Double quote	`0x0022`
`\\`	Backslash	`0x005C`

www.it-ebooks.info

Escape Sequence	Character Name	Unicode Encoding
\0	Null	0x0000
\a	Alert	0x0007
\b	Backspace	0x0008
\f	Form feed	0x000C
\n	New line	0x000A
\r	Carriage return	0x000D
\t	Horizontal tab	0x0009
\v	Vertical tab	0x000B

The type of a *character-literal* is `char`.

2.4.4.5 *String Literals*

C# supports two forms of string literals: ***regular string literals*** and ***verbatim string literals***.

A regular string literal consists of zero or more characters enclosed in double quotes, as in `"hello"`, and may include both simple escape sequences (such as `\t` for the tab character), and hexadecimal and Unicode escape sequences.

A verbatim string literal consists of an `@` character followed by a double-quote character, zero or more characters, and a closing double-quote character. A simple example is `@"hello"`. In a verbatim string literal, the characters between the delimiters are interpreted verbatim, the only exception being a *quote-escape-sequence*. In particular, simple escape sequences and hexadecimal and Unicode escape sequences are not processed in verbatim string literals. A verbatim string literal may span multiple lines.

JON SKEET One aspect of verbatim string literals that makes me nervous is the way a line break will take on the form in which it occurs in the file. It's the natural option of course, but it means switching a file between "`\r\n`" and "`\n`" line breaks isn't a purely decorative change: It can affect behavior, too.

www.it-ebooks.info

string-literal:
    regular-string-literal
    verbatim-string-literal

regular-string-literal:
    " *regular-string-literal-characters*$_{opt}$ "

regular-string-literal-characters:
    regular-string-literal-character
    regular-string-literal-characters regular-string-literal-character

regular-string-literal-character:
    single-regular-string-literal-character
    simple-escape-sequence
    hexadecimal-escape-sequence
    unicode-escape-sequence

single-regular-string-literal-character:
    Any character except " (`U+0022`), \ (`U+005C`), and *new-line-character*

verbatim-string-literal:
    `@"` *verbatim-string-literal-characters*$_{opt}$ "

verbatim-string-literal-characters:
    verbatim-string-literal-character
    verbatim-string-literal-characters verbatim-string-literal-character

verbatim-string-literal-character:
    single-verbatim-string-literal-character
    quote-escape-sequence

single-verbatim-string-literal-character:
    Any character except "

quote-escape-sequence:
    `""`

A character that follows a backslash character (\) in a *regular-string-literal-character* must be one of the following characters: `'`, `"`, `\`, `0`, `a`, `b`, `f`, `n`, `r`, `t`, `u`, `U`, `x`, `v`. Otherwise, a compile-time error occurs.

www.it-ebooks.info

The example

```
string a = "hello, world";                  // hello, world
string b = @"hello, world";                 // hello, world

string c = "hello \t world";                // hello     world
string d = @"hello \t world";               // hello \t world

string e = "Joe said \"Hello\" to me";      // Joe said "Hello" to me
string f = @"Joe said ""Hello"" to me";     // Joe said "Hello" to me

string g = "\\\\server\\share\\file.txt";   // \\server\share\file.txt
string h = @"\\server\share\file.txt";      // \\server\share\file.txt

string i = "one\r\ntwo\r\nthree";
string j = @"one
two
three";
```

shows a variety of string literals. The last string literal, `j`, is a verbatim string literal that spans multiple lines. The characters between the quotation marks, including white space such as new line characters, are preserved verbatim.

BRAD ABRAMS In the early days of the C# language design, we experimented with using the backtick (`) character. Personally, I really liked this choice: The backtick is in the quote family and is a grossly underused character. In fact, it was so underused that some international keyboards don't even include a key for it. We were lucky to have such a good, early international following for C#, which allowed us to fix this potential error very early in the design of the language.

Since a hexadecimal escape sequence can have a variable number of hex digits, the string literal `"\x123"` contains a single character with hex value 123. To create a string containing the character with hex value 12 followed by the character 3, one could write `"\x00123"` or `"\x12" + "3"` instead.

JON SKEET Hexadecimal escape sequences aren't just rare—they're dangerous. While a string such as `"\x9Tabbed"` is reasonably clear, it's not nearly as obvious that `"\x9Badly tabbed"` actually begins with the Unicode character `U+9BAD`.

The type of a *string-literal* is `string`.

Each string literal does not necessarily result in a new string instance. When two or more string literals that are equivalent according to the string equality operator (§7.10.7) appear

www.it-ebooks.info

in the same program, these string literals refer to the same string instance. For instance, the output produced by

```
class Test
{
    static void Main() {
        object a = "hello";
        object b = "hello";
        System.Console.WriteLine(a == b);
    }
}
```

is `True` because the two literals refer to the same string instance.

> **JOSEPH ALBAHARI** This optimization is called *interning*. One of the benefits of interning is that it reduces the size of the compiled assembly because duplicate string literals are factored out.
>
> In the preceding example, `a` and `b` are declared of type `object`, which forces the subsequent comparison to use `object`'s reference-type equality semantics. If `a` and `b` were declared of type `string`, the comparison would bind to `string`'s `==` operator, which evaluates to `true` if the strings have identical content—even if they refer to different underlying objects.

2.4.4.6 *The null Literal*

null-literal:
`null`

The *null-literal* can be implicitly converted to a reference type or nullable type.

> **ERIC LIPPERT** The null literal expression itself does not have a type.

2.4.5 Operators and Punctuators

There are several kinds of operators and punctuators. Operators are used in expressions to describe operations involving one or more operands. For example, the expression `a + b` uses the `+` operator to add the two operands `a` and `b`. Punctuators are for grouping and separating.

www.it-ebooks.info

operator-or-punctuator: one of

```
{      }      [      ]      (      )      .      ,      :      ;
+      -      *      /      %      &      |      ^      !      ~
=      <      >      ?      ??     ::     ++     --     &&     ||
->     ==     !=     <=     >=     +=     -=     *=     /=     %=
&=     |=     ^=     <<     <<=    =>
```

right-shift:

```
>|>
```

right-shift-assignment:

```
>|>=
```

The vertical bars in the *right-shift* and *right-shift-assignment* productions are used to indicate that, unlike other productions in the syntactic grammar, no characters of any kind (not even white space) are allowed between the tokens. These productions are treated specially to enable the correct handling of *type-parameter-list*s (§10.1.3).

2.5 Preprocessing Directives

BILL WAGNER This section shows how the C heritage colors so much of modern development. I rarely, if ever, use any of the preprocessing directives—but I can only imagine the storm of criticism if C# had not included a strong set of preprocessing directives.

The preprocessing directives provide the ability to conditionally skip sections of source files, to report error and warning conditions, and to delineate distinct regions of source code. The term "preprocessing directives" is used only for consistency with the C and C++ programming languages. In C#, there is no separate preprocessing step; preprocessing directives are processed as part of the lexical analysis phase.

pp-directive:
- *pp-declaration*
- *pp-conditional*
- *pp-line*
- *pp-diagnostic*
- *pp-region*
- *pp-pragma*

www.it-ebooks.info

The following preprocessing directives are available:

- `#define` and `#undef`, which are used to define and undefine, respectively, conditional compilation symbols (§2.5.3).
- `#if`, `#elif`, `#else`, and `#endif`, which are used to conditionally skip sections of source code (§2.5.4).
- `#line`, which is used to control line numbers emitted for errors and warnings (§2.5.7).
- `#error` and `#warning`, which are used to issue errors and warnings, respectively (§2.5.5).
- `#region` and `#endregion`, which are used to explicitly mark sections of source code (§2.5.6).
- `#pragma`, which is used to specify optional contextual information to the compiler (§2.5.8).

A preprocessing directive always occupies a separate line of source code and always begins with a `#` character and a preprocessing directive name. White space may occur before the `#` character and between the `#` character and the directive name.

A source line containing a `#define`, `#undef`, `#if`, `#elif`, `#else`, `#endif`, or `#line` directive may end with a single-line comment. Delimited comments (the `/* */` style of comments) are not permitted on source lines containing preprocessing directives.

Preprocessing directives are not tokens and are not part of the syntactic grammar of C#. However, preprocessing directives can be used to include or exclude sequences of tokens and can in that way affect the meaning of a C# program. For example, when compiled, the program

```
#define A
#undef B

class C
{
#if A
    void F() {}
#else
    void G() {}
#endif

#if B
    void H() {}
#else
    void I() {}
#endif
}
```


www.it-ebooks.info

results in the exact same sequence of tokens as the program

```
class C
{
    void F() {}
    void I() {}
}
```

Thus, whereas lexically the two programs are quite different, syntactically they are identical.

2.5.1 Conditional Compilation Symbols

The conditional compilation functionality provided by the `#if`, `#elif`, `#else`, and `#endif` directives is controlled through preprocessing expressions (§2.5.2) and conditional compilation symbols.

conditional-symbol:
    Any *identifier-or-keyword* except `true` or `false`

A conditional compilation symbol has two possible states: ***defined*** or ***undefined***. At the beginning of the lexical processing of a source file, a conditional compilation symbol is undefined unless it has been explicitly defined by an external mechanism (such as a command-line compiler option). When a `#define` directive is processed, the conditional compilation symbol named in that directive becomes defined in that source file. The symbol remains defined until an `#undef` directive for that same symbol is processed, or until the end of the source file is reached. An implication of this is that `#define` and `#undef` directives in one source file have no effect on other source files in the same program.

When referenced in a preprocessing expression, a defined conditional compilation symbol has the boolean value `true`, and an undefined conditional compilation symbol has the boolean value `false`. There is no requirement that conditional compilation symbols be explicitly declared before they are referenced in preprocessing expressions. Instead, undeclared symbols are simply undefined and thus have the value `false`.

The name space for conditional compilation symbols is distinct and separate from all other named entities in a C# program. Conditional compilation symbols can only be referenced in `#define` and `#undef` directives and in preprocessing expressions.

2.5.2 Preprocessing Expressions

Preprocessing expressions can occur in `#if` and `#elif` directives. The operators `!`, `==`, `!=`, `&&`, and `||` are permitted in preprocessing expressions, and parentheses may be used for grouping.

www.it-ebooks.info

pp-expression:
    whitespace$_{opt}$ *pp-or-expression* *whitespace*$_{opt}$

pp-or-expression:
    pp-and-expression
    pp-or-expression *whitespace*$_{opt}$ `||` *whitespace*$_{opt}$ *pp-and-expression*

pp-and-expression:
    pp-equality-expression
    pp-and-expression *whitespace*$_{opt}$ `&&` *whitespace*$_{opt}$ *pp-equality-expression*

pp-equality-expression:
    pp-unary-expression
    pp-equality-expression *whitespace*$_{opt}$ `==` *whitespace*$_{opt}$ *pp-unary-expression*
    pp-equality-expression *whitespace*$_{opt}$ `!=` *whitespace*$_{opt}$ *pp-unary-expression*

pp-unary-expression:
    pp-primary-expression
    `!` *whitespace*$_{opt}$ *pp-unary-expression*

pp-primary-expression:
    `true`
    `false`
    conditional-symbol
    `(` *whitespace*$_{opt}$ *pp-expression* *whitespace*$_{opt}$ `)`

When referenced in a preprocessing expression, a defined conditional compilation symbol has the boolean value `true`, and an undefined conditional compilation symbol has the boolean value `false`.

Evaluation of a preprocessing expression always yields a boolean value. The rules of evaluation for a preprocessing expression are the same as those for a constant expression (§7.19), except that the only user-defined entities that can be referenced are conditional compilation symbols.

2.5.3 Declaration Directives

The declaration directives are used to define or undefine conditional compilation symbols.

pp-declaration:
    whitespace$_{opt}$ `#` *whitespace*$_{opt}$ `define` *whitespace* *conditional-symbol* *pp-new-line*
    whitespace$_{opt}$ `#` *whitespace*$_{opt}$ `undef` *whitespace* *conditional-symbol* *pp-new-line*

pp-new-line:
    whitespace$_{opt}$ *single-line-comment*$_{opt}$ *new-line*

www.it-ebooks.info

The processing of a `#define` directive causes the given conditional compilation symbol to become defined, starting with the source line that follows the directive. Likewise, the processing of a `#undef` directive causes the given conditional compilation symbol to become undefined, starting with the source line that follows the directive.

Any `#define` and `#undef` directives in a source file must occur before the first *token* (§2.4) in the source file; otherwise, a compile-time error occurs. In intuitive terms, `#define` and `#undef` directives must precede any "real code" in the source file.

The example

```
#define Enterprise

#if Professional || Enterprise
    #define Advanced
#endif

namespace Megacorp.Data
{
    #if Advanced
    class PivotTable {...}
    #endif
}
```

is valid because the `#define` directives precede the first token (the `namespace` keyword) in the source file.

The following example results in a compile-time error because a `#define` follows real code:

```
#define A
namespace N
{
    #define B
    #if B
    class Class1 {}
    #endif
}
```

A `#define` may define a conditional compilation symbol that is already defined, without there being any intervening `#undef` for that symbol. The example below defines a conditional compilation symbol `A` and then defines it again.

```
#define A
#define A
```

A `#undef` may "undefine" a conditional compilation symbol that is not defined. The example below defines a conditional compilation symbol `A` and then undefines it twice; although the second `#undef` has no effect, it is still valid.

```
#define A
#undef A
#undef A
```


www.it-ebooks.info

2.5.4 Conditional Compilation Directives

The conditional compilation directives are used to conditionally include or exclude portions of a source file.

pp-conditional:
pp-if-section *pp-elif-sections*$_{opt}$ *pp-else-section*$_{opt}$ *pp-endif*

pp-if-section:
whitespace$_{opt}$ `#` *whitespace*$_{opt}$ `if` *whitespace* *pp-expression* *pp-new-line*
conditional-section$_{opt}$

pp-elif-sections:
pp-elif-section
pp-elif-sections *pp-elif-section*

pp-elif-section:
whitespace$_{opt}$ `#` *whitespace*$_{opt}$ `elif` *whitespace* *pp-expression* *pp-new-line*
conditional-section$_{opt}$

pp-else-section:
whitespace$_{opt}$ `#` *whitespace*$_{opt}$ `else` *pp-new-line* *conditional-section*$_{opt}$

pp-endif:
whitespace$_{opt}$ `#` *whitespace*$_{opt}$ `endif` *pp-new-line*

conditional-section:
input-section
skipped-section

skipped-section:
skipped-section-part
skipped-section *skipped-section-part*

skipped-section-part:
skipped-characters$_{opt}$ *new-line*
pp-directive

skipped-characters:
whitespace$_{opt}$ *not-number-sign* *input-characters*$_{opt}$

not-number-sign:
Any *input-character* except `#`

www.it-ebooks.info

As indicated by the syntax, conditional compilation directives must be written as sets consisting of, in order, an `#if` directive, zero or more `#elif` directives, zero or one `#else` directive, and an `#endif` directive. Between the directives are conditional sections of source code. Each section is controlled by the immediately preceding directive. A conditional section may itself contain nested conditional compilation directives provided these directives form complete sets.

> **CHRIS SELLS** I can see `#if` and `#endif`, but `#elif`? Is this a Santa's helper with a lisp? A hipster abbreviation for some large, gray, wrinkled animal? I'd have preferred springing for the two extra characters in `#elseif` simply so I could actually remember it. . .

A *pp-conditional* selects at most one of the contained *conditional-section*s for normal lexical processing:

- The *pp-expression*s of the `#if` and `#elif` directives are evaluated in order until one yields `true`. If an expression yields `true`, the *conditional-section* of the corresponding directive is selected.
- If all *pp-expression*s yield `false`, and if an `#else` directive is present, the *conditional-section* of the `#else` directive is selected.
- Otherwise, no *conditional-section* is selected.

The selected *conditional-section*, if any, is processed as a normal *input-section*: The source code contained in the section must adhere to the lexical grammar; tokens are generated from the source code in the section; and preprocessing directives in the section have the prescribed effects.

The remaining *conditional-section*s, if any, are processed as *skipped-section*s: Except for preprocessing directives, the source code in the section need not adhere to the lexical grammar; no tokens are generated from the source code in the section; and preprocessing directives in the section must be lexically correct but are not otherwise processed. Within a *conditional-section* that is being processed as a *skipped-section*, any nested *conditional-section*s (contained in nested `#if`...`#endif` and `#region`...`#endregion` constructs) are also processed as *skipped-section*s.

The following example illustrates how conditional compilation directives can nest:

```
#define Debug     // Debugging on
#undef Trace      // Tracing off

class PurchaseTransaction
{
```


www.it-ebooks.info

```
        void Commit() {
            #if Debug
                CheckConsistency();
                #if Trace
                    WriteToLog(this.ToString());
                #endif
            #endif
            CommitHelper();
        }
    }
```

Except for preprocessing directives, skipped source code is not subject to lexical analysis. For example, the following is valid despite the unterminated comment in the `#else` section:

```
#define Debug           // Debugging on

class PurchaseTransaction
{
    void Commit() {
        #if Debug
            CheckConsistency();
        #else
            // Do something else
        #endif
    }
}
```

Note, however, that preprocessing directives are required to be lexically correct even in skipped sections of source code.

CHRIS SELLS Avoid the use of nested preprocessing directives if you can, simply because by default the most popular C# editor on the planet, Visual Studio, will arrange them along the left edge of your text file, making it very difficult to follow the nesting. For example,

```
#define Debug    // Debugging on
#undef Trace     // Tracing off
class PurchaseTransaction {
    void Commit() {
#if Debug
        CheckConsistency();
#if Trace
        WriteToLog(this.ToString());
#endif
#endif
        CommitHelper();
    }
```


www.it-ebooks.info

Preprocessing directives are not processed when they appear inside multi-line input elements. For example, the program

```
class Hello
{
    static void Main() {
        System.Console.WriteLine(@"hello,
#if Debug
        World
#else
        Nebraska
#endif
        ");
    }
}
```

results in the output:

```
hello,
#if Debug
        world
#else
        Nebraska
#endif
```

In peculiar cases, the set of preprocessing directives that is processed might depend on the evaluation of the *pp-expression*. The example

```
#if X
    /*
#else
    /* */ class Q { }
#endif
```

always produces the same token stream (`class Q { }`), regardless of whether `X` is defined. If `X` is defined, the only processed directives are `#if` and `#endif`, due to the multi-line comment. If `X` is undefined, then three directives (`#if`, `#else`, `#endif`) are part of the directive set.

2.5.5 Diagnostic Directives

The diagnostic directives are used to explicitly generate error and warning messages that are reported in the same way as other compile-time errors and warnings.

pp-diagnostic:
whitespace$_{opt}$ `#` *whitespace*$_{opt}$ `error` *pp-message*
whitespace$_{opt}$ `#` *whitespace*$_{opt}$ `warning` *pp-message*

www.it-ebooks.info

pp-message:
    new-line
    whitespace *input-characters*$_{opt}$ *new-line*

The example

```
#warning Code review needed before check-in

#if Debug && Retail
    #error A build can't be both debug and retail
#endif

class Test {...}
```

always produces a warning ("Code review needed before check-in"), and produces a compile-time error ("A build can't be both debug and retail") if the conditional symbols `Debug` and `Retail` are both defined. Note that a *pp-message* can contain arbitrary text; specifically, it need not contain well-formed tokens, as shown by the single quote in the word `can't`.

2.5.6 Region Directives

The region directives are used to explicitly mark regions of source code.

pp-region:
    pp-start-region *conditional-section*$_{opt}$ *pp-end-region*

pp-start-region:
    whitespace$_{opt}$ `#` *whitespace*$_{opt}$ `region` *pp-message*

pp-end-region:
    whitespace$_{opt}$ `#` *whitespace*$_{opt}$ `endregion` *pp-message*

No semantic meaning is attached to a region; regions are intended for use by the programmer or by automated tools to mark a section of source code. The message specified in a `#region` or `#endregion` directive likewise has no semantic meaning; it merely serves to identify the region. Matching `#region` and `#endregion` directives may have different *pp-message*s.

The lexical processing of a region

```
#region
...
#endregion
```

corresponds exactly to the lexical processing of a conditional compilation directive of the form:

```
#if true
...
#endif
```


www.it-ebooks.info

2.5.7 Line Directives

Line directives may be used to alter the line numbers and source file names that are reported by the compiler in output such as warnings and errors.

Line directives are most commonly used in meta-programming tools that generate C# source code from some other text input.

pp-line:
    whitespace$_{opt}$ `#` *whitespace*$_{opt}$ `line` *whitespace* *line-indicator* *pp-new-line*

line-indicator:
    decimal-digits *whitespace* *file-name*
    decimal-digits
    `default`
    `hidden`

file-name:
    `"` *file-name-characters* `"`

file-name-characters:
    file-name-character
    file-name-characters *file-name-character*

file-name-character:
    Any *input-character* except "

When no `#line` directives are present, the compiler reports true line numbers and source file names in its output. When processing a `#line` directive that includes a *line-indicator* that is not `default`, the compiler treats the line *after* the directive as having the given line number (and file name, if specified).

JON SKEET If you're writing a tool that generates code from some other source, you might think that the easiest approach is to just include a `#line` directive for every line you output. It's not quite that simple, however: A `#line` directive within a verbatim string literal ends up as part of the string, rather than being treated as a directive. So the following code almost certainly doesn't work as intended:

```
#line 5
Console.WriteLine(@"First line
#line 6
Second line");
```


www.it-ebooks.info

A `#line default` directive reverses the effect of all preceding #line directives. The compiler reports true line information for subsequent lines, precisely as if no `#line` directives had been processed.

A `#line hidden` directive has no effect on the file and line numbers reported in error messages, but does affect source-level debugging. When debugging, all lines between a `#line hidden` directive and the subsequent `#line` directive (that is not `#line hidden`) have no line number information. When stepping through code in the debugger, these lines will be skipped entirely.

Note that a *file-name* differs from a regular string literal in that escape characters are not processed; the '\' character simply designates an ordinary backslash character within a *file-name*.

2.5.8 Pragma Directives

The `#pragma` preprocessing directive is used to specify optional contextual information to the compiler. The information supplied in a `#pragma` directive will never change program semantics.

pp-pragma:
    whitespace$_{opt}$ `#` *whitespace*$_{opt}$ `pragma` *whitespace* *pragma-body* *pp-new-line*

pragma-body:
    pragma-warning-body

C# provides `#pragma` directives to control compiler warnings. Future versions of the language may include additional `#pragma` directives. To ensure interoperability with other C# compilers, the Microsoft C# compiler does not issue compilation errors for unknown `#pragma` directives; such directives do, however, generate warnings.

2.5.8.1 *Pragma Warning*

The `#pragma warning` directive is used to disable or restore all or a particular set of warning messages during compilation of the subsequent program text.

pragma-warning-body:
    `warning` *whitespace* *warning-action*
    `warning` *whitespace* *warning-action* *whitespace* *warning-list*

warning-action:
    `disable`
    `restore`

www.it-ebooks.info

warning-list:
    decimal-digits
    warning-list *whitespace*$_{opt}$, *whitespace*$_{opt}$ *decimal-digits*

A `#pragma warning` directive that omits the warning list affects all warnings. A `#pragma warning` directive that includes a warning list affects only those warnings that are specified in the list.

JON SKEET I'm not sure that defaulting to affecting all warnings is a good idea. Requiring an explicit "all" as an alternative warning list wouldn't have conflicted with any other warnings (because individual warnings have to be numeric) and it would make the meaning a good deal clearer. Having said this, I've never seen anyone disable all warnings—and I hope never to do so.

A `#pragma warning disable` directive disables all or the given set of warnings.

A `#pragma warning restore` directive restores all or the given set of warnings to the state that was in effect at the beginning of the compilation unit. Note that if a particular warning was disabled externally, a `#pragma warning restore` (whether for all or the specific warning) will not re-enable that warning.

The following example shows use of `#pragma warning` to temporarily disable the warning reported when obsoleted members are referenced, using the warning number from the Microsoft C# compiler.

```
using System;

class Program
{
    [Obsolete]
    static void Foo() {}

    static void Main() {
#pragma warning disable 612
    Foo();
#pragma warning restore 612
    }
}
```

JOSEPH ALBAHARI The compiler generates a warning when it detects a condition that might possibly indicate a mistake in the code. Because of the potential for false positives, the ability to disable certain warnings over selected lines of code is important in maintaining a good signal-to-noise ratio—so that the real mistakes are noticed. It's also essential if you want to instruct the compiler to treat warnings as errors—something I do in my own projects

www.it-ebooks.info

This page intentionally left blank

www.it-ebooks.info

3. Basic Concepts

3.1 Application Start-up

An assembly that has an ***entry point*** is called an ***application***. When an application is run, a new ***application domain*** is created. Several different instantiations of an application may exist on the same machine at the same time, and each has its own application domain.

An application domain enables application isolation by acting as a container for application state. An application domain acts as a container and boundary for the types defined in the application and the class libraries it uses. Types loaded into one application domain are distinct from the same type loaded into another application domain, and instances of objects are not directly shared between application domains. For instance, each application domain has its own copy of static variables for these types, and a static constructor for a type is run at most once per application domain. Implementations are free to provide implementation-specific policy or mechanisms for the creation and destruction of application domains.

Application start-up occurs when the execution environment calls a designated method, which is referred to as the application's entry point. This entry point method is always named `Main`, and can have one of the following signatures:

```
static void Main() {...}

static void Main(string[] args) {... }

static int Main() {...}

static int Main(string [] args) {...}
```

As shown, the entry point may optionally return an `int` value. This return value is used in application termination (§3.2).

The entry point may optionally have one formal parameter. The parameter may have any name, but the type of the parameter must be `string[]`. If the formal parameter is present, the execution environment creates and passes a `string[]` argument containing the command-line arguments that were specified when the application was started. The `string[]` argument is never null, but it may have a length of zero if no command-line arguments were specified.

www.it-ebooks.info

Since C# supports method overloading, a class or struct may contain multiple definitions of some method, provided each has a different signature. However, within a single program, no class or struct may contain more than one method called `Main` whose definition qualifies it to be used as an application entry point. Other overloaded versions of `Main` are permitted, however, provided they have more than one parameter, or their only parameter is other than type `string[]`.

An application can be made up of multiple classes or structs. It is possible for more than one of these classes or structs to contain a method called `Main` whose definition qualifies it to be used as an application entry point. In such cases, an external mechanism (such as a command-line compiler option) must be used to select one of these `Main` methods as the entry point.

> **ERIC LIPPERT** The "csc" command-line compiler provides the `/main:` switch for this purpose.

In C#, every method must be defined as a member of a class or struct. Ordinarily, the declared accessibility (§3.5.1) of a method is determined by the access modifiers (§10.3.5) specified in its declaration, and similarly the declared accessibility of a type is determined by the access modifiers specified in its declaration. For a given method of a given type to be callable, both the type and the member must be accessible. However, the application entry point is a special case. Specifically, the execution environment can access the application's entry point regardless of its declared accessibility and regardless of the declared accessibility of its enclosing type declarations.

The application entry point method may not be in a generic class declaration.

In all other respects, entry point methods behave like those that are not entry points.

3.2 Application Termination

Application termination returns control to the execution environment.

If the return type of the application's ***entry point*** method is `int`, the value returned serves as the application's ***termination status code***. The purpose of this code is to allow communication of success or failure to the execution environment.

If the return type of the entry point method is `void`, reaching the right brace (`}`) that terminates the method, or executing a `return` statement that has no expression, results in a termination status code of `0`.

www.it-ebooks.info

BILL WAGNER The following rule is an important difference between C# and other managed environments.

Prior to an application's termination, destructors for all of its objects that have not yet been garbage collected are called, unless such cleanup has been suppressed (by a call to the library method `GC.SuppressFinalize`, for example).

3.3 Declarations

Declarations in a C# program define the constituent elements of the program. C# programs are organized using namespaces (§9), which can contain type declarations and nested namespace declarations. Type declarations (§9.6) are used to define classes (§10), structs (§10.14), interfaces (§13), enums (§14), and delegates (§15). The kinds of members permitted in a type declaration depend on the form of the type declaration. For instance, class declarations can contain declarations for constants (§10.4), fields (§10.5), methods (§10.6), properties (§10.7), events (§10.8), indexers (§10.9), operators (§10.10), instance constructors (§10.11), static constructors (§10.12), destructors (§10.13), and nested types (§10.3.8).

A declaration defines a name in the ***declaration space*** to which the declaration belongs. Except for overloaded members (§3.6), it is a compile-time error to have two or more declarations that introduce members with the same name in a declaration space. It is never possible for a declaration space to contain different kinds of members with the same name. For example, a declaration space can never contain a field and a method by the same name.

ERIC LIPPERT "Declaration spaces" are frequently confused with "scopes." Although related conceptually, the two have quite different purposes. The scope of a named element is the region of the program text in which that element may be referred to by name without additional qualification. By contrast, the declaration space of an element is the region in which no two elements may have the same name. (Well, almost—methods may have the same name if they differ in signature, and types may have the same name if they differ in generic arity.)

There are several different types of declaration spaces.

- Within all source files of a program, *namespace-member-declaration*s with no enclosing *namespace-declaration* are members of a single combined declaration space called the ***global declaration space***.

www.it-ebooks.info

- Within all source files of a program, *namespace-member-declaration*s within *namespace-declaration*s that have the same fully qualified namespace name are members of a single combined declaration space.
- Each class, struct, or interface declaration creates a new declaration space. Names are introduced into this declaration space through *class-member-declaration*s, *struct-member-declaration*s, *interface-member-declaration*s, or *type-parameter*s. Except for overloaded instance constructor declarations and static constructor declarations, a class or struct cannot contain a member declaration with the same name as the class or struct. A class, struct, or interface permits the declaration of overloaded methods and indexers. Furthermore, a class or struct permits the declaration of overloaded instance constructors and operators. For example, a class, struct, or interface may contain multiple method declarations with the same name, provided these method declarations differ in their signature (§3.6). Note that base classes do not contribute to the declaration space of a class, and base interfaces do not contribute to the declaration space of an interface. Thus a derived class or interface is allowed to declare a member with the same name as an inherited member. Such a member is said to ***hide*** the inherited member.
- Each delegate declaration creates a new declaration space. Names are introduced into this declaration space through formal parameters (*fixed-parameter*s and *parameter-array*s) and *type-parameter*s.
- Each enumeration declaration creates a new declaration space. Names are introduced into this declaration space through *enum-member-declarations*.
- Each method declaration, indexer declaration, operator declaration, instance constructor declaration, and anonymous function creates a new declaration space called a ***local variable declaration space***. Names are introduced into this declaration space through formal parameters (*fixed-parameter*s and *parameter-array*s) and *type-parameter*s. The body of the function member or anonymous function, if any, is considered to be nested within the local variable declaration space. It is an error for a local variable declaration space and a nested local variable declaration space to contain elements with the same name. Thus, within a nested declaration space, it is not possible to declare a local variable or constant with the same name as a local variable or constant in an enclosing declaration space. It is possible for two declaration spaces to contain elements with the same name as long as neither declaration space contains the other.
- Each *block* or *switch-block*, as well as each *for*, *foreach*, and *using* statement, creates a local variable declaration space for local variables and local constants. Names are introduced into this declaration space through *local-variable-declaration*s and *local-constant-declaration*s. Note that blocks that occur as or within the body of a function member or anonymous function are nested within the local variable declaration space declared by

www.it-ebooks.info

those functions for their parameters. Thus it is an error to have, for example, a method with a local variable and a parameter of the same name.

- Each *block* or *switch-block* creates a separate declaration space for labels. Names are introduced into this declaration space through *labeled-statement*s, and the names are referenced through *goto-statement*s. The ***label declaration space*** of a block includes any nested blocks. Thus, within a nested block, it is not possible to declare a label with the same name as a label in an enclosing block.

The textual order in which names are declared is generally of no significance. In particular, textual order is not significant for the declaration and use of namespaces, constants, methods, properties, events, indexers, operators, instance constructors, destructors, static constructors, and types. Declaration order is significant in the following ways:

- Declaration order for field declarations and local variable declarations determines the order in which their initializers (if any) are executed.
- Local variables must be defined before they are used (§3.7).
- Declaration order for enum member declarations (§14.3) is significant when *constant-expression* values are omitted.

The declaration space of a namespace is "open-ended," and two namespace declarations with the same fully qualified name contribute to the same declaration space. For example:

```
namespace Megacorp.Data
{
    class Customer
    {
        ...
    }
}

namespace Megacorp.Data
{
    class Order
    {
        ...
    }
}
```

The two namespace declarations above contribute to the same declaration space, in this case declaring two classes with the fully qualified names `Megacorp.Data.Customer` and `Megacorp.Data.Order`. Because the two declarations contribute to the same declaration space, it would have caused a compile-time error if each contained a declaration of a class with the same name.

www.it-ebooks.info

> **BILL WAGNER** Think of namespaces as a tool to manage the logical organization of your code. By comparison, assemblies manage the physical organization of your code.

As specified above, the declaration space of a block includes any nested blocks. Thus, in the following example, the `F` and `G` methods result in a compile-time error because the name `i` is declared in the outer block and cannot be redeclared in the inner block. However, the `H` and `I` methods are valid because the two `i`'s are declared in separate non-nested blocks.

```
class A
    {
        void F()
        {
            int i = 0;
            if (true)
            {
                int i = 1;
            }
        }

        void G()
        {
            if (true)
            {
                int i = 0;
            }
            int i = 1;
        }

        void H()
        {
            if (true)
            {
                int i = 0;
            }
            if (true)
            {
                int i = 1;
            }
        }

        void I()
        {
            for (int i = 0; i < 10; i++)
                H();
            for (int i = 0; i < 10; i++)
                H();
        }
    }
```


www.it-ebooks.info

3.4 Members

Namespaces and types have ***members***. The members of an entity are generally available through the use of a qualified name that starts with a reference to the entity, followed by a "`.`" token, followed by the name of the member.

Members of a type are either declared in the type declaration or ***inherited*** from the base class of the type. When a type inherits from a base class, all members of the base class, except instance constructors, destructors, and static constructors, become members of the derived type. The declared accessibility of a base class member does not control whether the member is inherited—inheritance extends to any member that isn't an instance constructor, static constructor, or destructor. However, an inherited member may not be accessible in a derived type, either because of its declared accessibility (§3.5.1) or because it is hidden by a declaration in the type itself (§3.7.1.2).

3.4.1 Namespace Members

Namespaces and types that have no enclosing namespace are members of the ***global namespace***. This corresponds directly to the names declared in the global declaration space.

Namespaces and types declared within a namespace are members of that namespace. This corresponds directly to the names declared in the declaration space of the namespace.

Namespaces have no access restrictions. It is not possible to declare private, protected, or internal namespaces, and namespace names are always publicly accessible.

3.4.2 Struct Members

The members of a struct are the members declared in the struct and the members inherited from the struct's direct base class `System.ValueType` and the indirect base class `object`.

The members of a simple type correspond directly to the members of the struct type aliased by the simple type:

- The members of `sbyte` are the members of the `System.SByte` struct.
- The members of `byte` are the members of the `System.Byte` struct.
- The members of `short` are the members of the `System.Int16` struct.
- The members of `ushort` are the members of the `System.UInt16` struct.
- The members of `int` are the members of the `System.Int32` struct.
- The members of `uint` are the members of the `System.UInt32` struct.
- The members of `long` are the members of the `System.Int64` struct.

www.it-ebooks.info

- The members of `ulong` are the members of the `System.UInt64` struct.
- The members of `char` are the members of the `System.Char` struct.
- The members of `float` are the members of the `System.Single` struct.
- The members of `double` are the members of the `System.Double` struct.
- The members of `decimal` are the members of the `System.Decimal` struct.
- The members of `bool` are the members of the `System.Boolean` struct.

3.4.3 Enumeration Members

The members of an enumeration are the constants declared in the enumeration and the members inherited from the enumeration's direct base class `System.Enum` and the indirect base classes `System.ValueType` and `object`.

3.4.4 Class Members

The members of a class are the members declared in the class and the members inherited from the base class (except for class `object`, which has no base class). The members inherited from the base class include the constants, fields, methods, properties, events, indexers, operators, and types of the base class, but not the instance constructors, destructors, and static constructors of the base class. Base class members are inherited without regard to their accessibility.

A class declaration may contain declarations of constants, fields, methods, properties, events, indexers, operators, instance constructors, destructors, static constructors, and types.

The members of `object` and `string` correspond directly to the members of the class types they alias:

- The members of `object` are the members of the `System.Object` class.
- The members of `string` are the members of the `System.String` class.

3.4.5 Interface Members

The members of an interface are the members declared in the interface and in all base interfaces of the interface. The members in class `object` are not, strictly speaking, members of any interface (§13.2). However, the members in class `object` are available via member lookup in any interface type (§7.4).

3.4.6 Array Members

The members of an array are the members inherited from class `System.Array`.

www.it-ebooks.info

3.4.7 Delegate Members

The members of a delegate are the members inherited from class `System.Delegate`.

> **VLADIMIR RESHETNIKOV** In the Microsoft implementation of C#, the members of a delegate also include the instance methods `Invoke`, `BeginInvoke`, and `EndInvoke`, and the members inherited from class `System.MulticastDelegate`.

> **JON SKEET** The `Invoke`, `BeginInvoke`, and `EndInvoke` methods mentioned by Vladimir cannot be specified within the `Delegate` or `MulticastDelegate` types, as they depend on the parameters and return type of the delegate. This is an example of parameterized typing that generics couldn't quite handle even if it had been present from the first version of C#.

3.5 Member Access

Declarations of members allow control over member access. The accessibility of a member is established by the declared accessibility (§3.5.1) of the member combined with the accessibility of the immediately containing type, if any.

When access to a particular member is allowed, the member is said to be ***accessible***. Conversely, when access to a particular member is disallowed, the member is said to be ***inaccessible***. Access to a member is permitted when the textual location in which the access takes place is included in the accessibility domain (§3.5.2) of the member.

3.5.1 Declared Accessibility

The ***declared accessibility*** of a member can be one of the following:

- Public, which is selected by including a `public` modifier in the member declaration. The intuitive meaning of `public` is "access not limited."
- Protected, which is selected by including a `protected` modifier in the member declaration. The intuitive meaning of `protected` is "access limited to the containing class or types derived from the containing class."
- Internal, which is selected by including an `internal` modifier in the member declaration. The intuitive meaning of `internal` is "access limited to this program."
- Protected internal (meaning protected or internal), which is selected by including both a `protected` and an `internal` modifier in the member declaration. The intuitive meaning

www.it-ebooks.info

of `protected internal` is "access limited to this program or types derived from the containing class."

- Private, which is selected by including a `private` modifier in the member declaration. The intuitive meaning of `private` is "access limited to the containing type."

> **JESSE LIBERTY** While there is always a default accessibility, it is good programming practice to declare the accessibility explicitly. This makes for code that is easier to read and far easier to maintain.

> **JON SKEET** The default accessibility is nicely chosen in C#: It's always the most restrictive level available, with the exception of making a property getter/setter more restrictive than the overall property declaration. I used to prefer to leave the accessibility as an implicit value, but I've come around to Jesse's point of view over time. Making anything explicit indicates that you are aware that a choice exists, and that you have deliberately chosen this particular option. If you leave the choice implicit, it could be because you wanted that option—or it could be because you forgot there was a choice to make in the first place.

Depending on the context in which a member declaration takes place, only certain types of declared accessibility are permitted. Furthermore, when a member declaration does not include any access modifiers, the context in which the declaration takes place determines the default declared accessibility.

- Namespaces implicitly have `public` declared accessibility. No access modifiers are allowed on namespace declarations.
- Types declared in compilation units or namespaces can have `public` or `internal` declared accessibility and default to `internal` declared accessibility.
- Class members can have any of the five kinds of declared accessibility and default to `private` declared accessibility. (Note that a type declared as a member of a class can have any of the five kinds of declared accessibility, whereas a type declared as a member of a namespace can have only `public` or `internal` declared accessibility.)

> **VLADIMIR RESHETNIKOV** If a sealed class declares a `protected` or `protected internal` member, a warning is issued. If a static class declares a `protected` or `protected internal` member, a compile-time error occurs (CS1057).

www.it-ebooks.info

- Struct members can have `public`, `internal`, or `private` declared accessibility and default to `private` declared accessibility because structs are implicitly sealed. Struct members introduced in a struct (that is, not inherited by that struct) cannot have `protected` or `protected internal` declared accessibility. (Note that a type declared as a member of a struct can have `public`, `internal`, or `private` declared accessibility, whereas a type declared as a member of a namespace can have only `public` or `internal` declared accessibility.)
- Interface members implicitly have `public` declared accessibility. No access modifiers are allowed on interface member declarations.
- Enumeration members implicitly have `public` declared accessibility. No access modifiers are allowed on enumeration member declarations.

VLADIMIR RESHETNIKOV The wording "enumeration members" here means "members, declared in an enumeration." Enumerations also inherit members from their base classes `System.Enum`, `System.ValueType`, and `System.Object`, and those members can be non-public.

JOSEPH ALBAHARI The rationale behind these rules is that the default declared accessibility for any construct is the minimum accessibility that it requires to be useful. Minimizing accessibility is positive in the sense that it promotes encapsulation.

JESSE LIBERTY That said, it is good programming practice to make the accessibility explicit, which makes your code much easier to maintain.

ERIC LIPPERT There is a difference between the "declared" accessibility and the actual effective accessibility. For example, a method declared as `public` on a class declared as `internal` is, for most practical purposes, an internal method.

A good way to think about this issue is to recognize that a `public` class member is `public` only to the entities that have access to the class.

JON SKEET One notable exception to Eric's annotation is when you're overriding a `public` method within an `internal` (or even `private`) class—including implementing an interface. Many interface implementations will be within `internal` classes, but instances may still be available to other assemblies via the interface.

www.it-ebooks.info

3.5.2 Accessibility Domains

The ***accessibility domain*** of a member consists of the (possibly disjoint) sections of program text in which access to the member is permitted. For purposes of defining the accessibility domain of a member, a member is said to be ***top-level*** if it is not declared within a type, and a member is said to be ***nested*** if it is declared within another type. Furthermore, the ***program text*** of a program is defined as all program text contained in all source files of the program, and the program text of a type is defined as all program text contained in the *type-declaration*s of that type (including, possibly, types that are nested within the type).

The accessibility domain of a predefined type (such as `object`, `int`, or `double`) is unlimited.

The accessibility domain of a top-level unbound type `T` (§4.4.3) that is declared in a program `P` is defined as follows:

- If the declared accessibility of `T` is `public`, the accessibility domain of `T` is the program text of `P` and any program that references `P`.
- If the declared accessibility of `T` is `internal`, the accessibility domain of `T` is the program text of `P`.

From these definitions, it follows that the accessibility domain of a top-level unbound type is always at least the program text of the program in which that type is declared.

The accessibility domain for a constructed type $T<A_1, \ldots, A_N>$ is the intersection of the accessibility domain of the unbound generic type `T` and the accessibility domains of the type arguments $A_1, \ldots, A_N$.

The accessibility domain of a nested member `M` declared in a type `T` within a program `P` is defined as follows (noting that `M` itself may possibly be a type):

- If the declared accessibility of `M` is `public`, the accessibility domain of `M` is the accessibility domain of `T`.
- If the declared accessibility of `M` is `protected internal`, let `D` be the union of the program text of `P` and the program text of any type derived from `T`, which is declared outside `P`. The accessibility domain of `M` is the intersection of the accessibility domain of `T` with `D`.
- If the declared accessibility of `M` is `protected`, let `D` be the union of the program text of `T` and the program text of any type derived from `T`. The accessibility domain of `M` is the intersection of the accessibility domain of `T` with `D`.
- If the declared accessibility of `M` is `internal`, the accessibility domain of `M` is the intersection of the accessibility domain of `T` with the program text of `P`.
- If the declared accessibility of `M` is `private`, the accessibility domain of `M` is the program text of `T`.

www.it-ebooks.info

From these definitions, it follows that the accessibility domain of a nested member is always at least the program text of the type in which the member is declared. Furthermore, it follows that the accessibility domain of a member is never more inclusive than the accessibility domain of the type in which the member is declared.

In intuitive terms, when a type or member `M` is accessed, the following steps are evaluated to ensure that the access is permitted:

- First, if `M` is declared within a type (as opposed to a compilation unit or a namespace), a compile-time error occurs if that type is not accessible.
- Then, if `M` is `public`, the access is permitted.
- Otherwise, if `M` is `protected internal`, the access is permitted if it occurs within the program in which `M` is declared, or if it occurs within a class derived from the class in which `M` is declared and takes place through the derived class type (§3.5.3).
- Otherwise, if `M` is `protected`, the access is permitted if it occurs within the class in which `M` is declared, or if it occurs within a class derived from the class in which `M` is declared and takes place through the derived class type (§3.5.3).
- Otherwise, if `M` is `internal`, the access is permitted if it occurs within the program in which `M` is declared.
- Otherwise, if `M` is `private`, the access is permitted if it occurs within the type in which `M` is declared.
- Otherwise, the type or member is inaccessible, and a compile-time error occurs.

In the example

```
public class A
 {
     public static int X;
     internal static int Y;
     private static int Z;
 }

 internal class B
 {
     public static int X;
     internal static int Y;
     private static int Z;

     public class C
     {
         public static int X;
         internal static int Y;
         private static int Z;
     }

     private class D
```


www.it-ebooks.info

```
        {
            public static int X;
            internal static int Y;
            private static int Z;
        }
    }
```

the classes and members have the following accessibility domains:

- The accessibility domain of `A` and `A.X` is unlimited.
- The accessibility domain of `A.Y`, `B`, `B.X`, `B.Y`, `B.C`, `B.C.X`, and `B.C.Y` is the program text of the containing program.
- The accessibility domain of `A.Z` is the program text of `A`.
- The accessibility domain of `B.Z` and `B.D` is the program text of `B`, including the program text of `B.C` and `B.D`.
- The accessibility domain of `B.C.Z` is the program text of `B.C`.
- The accessibility domain of `B.D.X` and `B.D.Y` is the program text of `B`, including the program text of `B.C` and `B.D`.
- The accessibility domain of `B.D.Z` is the program text of `B.D`.

As the example illustrates, the accessibility domain of a member is never larger than that of a containing type. For example, even though all `X` members have public declared accessibility, all but `A.X` have accessibility domains that are constrained by a containing type.

JOSEPH ALBAHARI Declaring a `public` member within an `internal` type might seem pointless, given that the member's visibility will be capped at `internal`. It can make sense, however, if the `public` member modifier is interpreted as meaning "having the same visibility as the containing type."

A good question to ask in deciding whether to declare a member of an internal type as `public` or `internal` is this: If the type was later promoted to `public`, would I want this member to become `public`, too? If the answer is yes, one could argue for declaring the member as `public` from the outset.

JESSE LIBERTY While there are rational examples of each of the cases mentioned above, good programming practice strongly favors using the least complex and most obvious accessibility, to reduce confusion and make for more easily maintainable code. I've written hundreds of commercially viable applications using nothing more than `public`, `private`, and `protected`.

www.it-ebooks.info

As described in §3.4, all members of a base class, except for instance constructors, destructors, and static constructors, are inherited by derived types. This includes even `private` members of a base class. However, the accessibility domain of a `private` member includes only the program text of the type in which the member is declared. In the example

```
class A
{
    int x;

    static void F(B b)
    {
        b.x = 1;        // Ok
    }
}

class B : A
{
    static void F(B b)
    {
        b.x = 1;        // Error: x not accessible
    }
}
```

the `B` class inherits the `private` member `x` from the `A` class. Because the member is `private`, it is only accessible within the *class-body* of `A`. Thus the access to `b.x` succeeds in the `A.F` method, but fails in the `B.F` method.

BILL WAGNER Notice that the inaccessible methods also obviate the need for the new modifier on `B:F()`.

3.5.3 Protected Access for Instance Members

When a `protected` instance member is accessed outside the program text of the class in which it is declared, and when a `protected internal` instance member is accessed outside the program text of the program in which it is declared, the access must take place within a class declaration that derives from the class in which it is declared. Furthermore, the access is required to take place *through* an instance of that derived class type or a class type constructed from it. This restriction prevents one derived class from accessing `protected` members of other derived classes, even when the members are inherited from the same base class.

www.it-ebooks.info

ERIC LIPPERT For instance, suppose you have a base class `Animal` and two derived classes, `Mammal` and `Reptile`, and `Animal` has a protected method `Feed()`. Then the `Mammal` code can call `Feed()` on a `Mammal` or any subclass of `Mammal` (`Tiger`, say).

`Mammal` code cannot call `Feed()` on an expression of type `Reptile` because there is no inheritance relationship between `Mammal` and `Reptile`.

Furthermore, because an expression of type `Animal` might actually be a `Reptile` at runtime, `Mammal` code also cannot call `Feed()` on an expression of type `Animal`.

Let `B` be a base class that declares a `protected` instance member `M`, and let `D` be a class that derives from `B`. Within the *class-body* of `D`, access to `M` can take one of the following forms:

- An unqualified *type-name* or *primary-expression* of the form `M`.
- A *primary-expression* of the form `E.M`, provided the type of `E` is `T` or a class derived from `T`, where `T` is the class type `D`, or a class type constructed from `D`.
- A *primary-expression* of the form `base.M`.

In addition to these forms of access, a derived class can access a `protected` instance constructor of a base class in a *constructor-initializer* (§10.11.1).

VLADIMIR RESHETNIKOV Put simply, other forms of access are not allowed. For example, a derived class cannot invoke its base class's `protected` constructor in a `new` operator:

```
class Base
{
    protected Base() { }
}

class Derived : Base
{
    static void Main()
    {
        new Base();     // Error CS0122: 'Base.Base()' is
                        // inaccessible due to its
                        // protection level
    }
}
```


www.it-ebooks.info

In the example

```
public class A
{
    protected int x;

    static void F(A a, B b)
    {
        a.x = 1;        // Okay
        b.x = 1;        // Okay
    }
}

public class B : A
{
    static void F(A a, B b)
    {
        a.x = 1;        // Error: must access through instance of B
        b.x = 1;        // Okay
    }
}
```

within `A`, it is possible to access `x` through instances of both `A` and `B`, since in either case the access takes place *through* an instance of `A` or a class derived from `A`. However, within `B`, it is not possible to access `x` through an instance of `A`, since `A` does not derive from `B`.

In the example

```
class C<T>
{
    protected T x;
}

class D<T> : C<T>
{
    static void F()
    {
        D<T> dt = new D<T>();
        D<int> di = new D<int>();
        D<string> ds = new D<string>();
        dt.x = default(T);
        di.x = 123;
        ds.x = "test";
    }
}
```

the three assignments to `x` are permitted because they all take place through instances of class types constructed from the generic type.

www.it-ebooks.info

CHRIS SELLS To minimize the surface area of a class or namespace, I recommend keeping things `private/internal` until they prove to be necessary in a wider scope. Refactoring is your friend here.

BILL WAGNER These rules exist to allow for the separate evolution of components in different assemblies. You should not incorporate these rules into your regular design patterns.

3.5.4 Accessibility Constraints

Several constructs in the C# language require a type to be ***at least as accessible as*** a member or another type. A type `T` is said to be at least as accessible as a member or type `M` if the accessibility domain of `T` is a superset of the accessibility domain of `M`. In other words, `T` is at least as accessible as `M` if `T` is accessible in all contexts in which `M` is accessible.

VLADIMIR RESHETNIKOV For the purposes of this paragraph, only accessibility modifiers are considered. For instance, if a `protected` member is declared in a `public sealed` class, the fact that this class cannot have descendants (and, therefore, the accessibility domain of this member does not include any descendants) is not considered.

The following accessibility constraints exist:

- The direct base class of a class type must be at least as accessible as the class type itself.
- The explicit base interfaces of an interface type must be at least as accessible as the interface type itself.
- The return type and parameter types of a delegate type must be at least as accessible as the delegate type itself.
- The type of a constant must be at least as accessible as the constant itself.
- The type of a field must be at least as accessible as the field itself.
- The return type and parameter types of a method must be at least as accessible as the method itself.
- The type of a property must be at least as accessible as the property itself.
- The type of an event must be at least as accessible as the event itself.

www.it-ebooks.info

- The type and parameter types of an indexer must be at least as accessible as the indexer itself.
- The return type and parameter types of an operator must be at least as accessible as the operator itself.
- The parameter types of an instance constructor must be at least as accessible as the instance constructor itself.

In the example

```
class A {...}

public class B: A {...}
```

the `B` class results in a compile-time error because `A` is not at least as accessible as `B`.

Likewise, in the example

```
class A {...}

public class B
{
    A F() {...}

    internal A G() {...}

    public A H() {...}
}
```

the `H` method in `B` results in a compile-time error because the return type `A` is not at least as accessible as the method.

3.6 Signatures and Overloading

Methods, instance constructors, indexers, and operators are characterized by their ***signatures***:

- The signature of a method consists of the name of the method, the number of type parameters, and the type and kind (value, reference, or output) of each of its formal parameters, considered in the order left to right. For these purposes, any type parameter of the method that occurs in the type of a formal parameter is identified not by its name, but rather by its ordinal position in the type argument list of the method. The signature of a method specifically does not include the return type, the `params` modifier that may be specified for the rightmost parameter, or the optional type parameter constraints.

www.it-ebooks.info

JON SKEET The use of overloading by number of type parameters is interesting—and occurs at the type level, too (consider, for example, the .NET types `System.Nullable` and `System.Nullable<T>`). The fact that constraints aren't taken into account can very occasionally be irritating. In some cases, it could be handy to be able to write

```
void Process<T>(T reference) where T : class { ... }
void Process<T>(T value) where T : struct { ... }
```

but this would introduce more complexity into an area that can already be very tricky to reason about. Developers' minds have been known to spontaneously combust when overloading and type inference meet.

- The signature of an instance constructor consists of the type and kind (value, reference, or output) of each of its formal parameters, considered in the order left to right. The signature of an instance constructor specifically does not include the `params` modifier that may be specified for the rightmost parameter.
- The signature of an indexer consists of the type of each of its formal parameters, considered in the order left to right. The signature of an indexer specifically does not include the element type, nor does it include the `params` modifier that may be specified for the rightmost parameter.
- The signature of an operator consists of the name of the operator and the type of each of its formal parameters, considered in the order left to right. The signature of an operator specifically does not include the result type.

Signatures are the enabling mechanism for ***overloading*** of members in classes, structs, and interfaces:

- Overloading of methods permits a class, struct, or interface to declare multiple methods with the same name, provided their signatures are unique within that class, struct, or interface.
- Overloading of instance constructors permits a class or struct to declare multiple instance constructors, provided their signatures are unique within that class or struct.
- Overloading of indexers permits a class, struct, or interface to declare multiple indexers, provided their signatures are unique within that class, struct, or interface.

www.it-ebooks.info

VLADIMIR RESHETNIKOV One exception to the uniqueness requirement for indexer signatures: If an indexer is an explicit implementation of an interface, its signature is checked for uniqueness only within those interface's explicitly implemented indexers:

```
class A : IIndexable
{
    int IIndexable.this[int x]
    {
        get { /* ... */ }
    }

    public int this[int x] // Okay
    {
        get { /* ... */ }
    }
}
```

- Overloading of operators permits a class or struct to declare multiple operators with the same name, provided their signatures are unique within that class or struct.

Although `out` and `ref` parameter modifiers are considered part of a signature, members declared in a single type cannot differ in signature solely by `ref` and `out`. A compile-time error occurs if two members are declared in the same type with signatures that would be the same if all parameters in both methods with `out` modifiers were changed to `ref` modifiers. For other purposes of signature matching (e.g., hiding or overriding), `ref` and `out` are considered part of the signature and do not match each other. (This restriction is to allow C# programs to be easily translated to run on the Common Language Infrastructure [CLI], which does not provide a way to define methods that differ solely in `ref` and `out`.)

For the purposes of signatures, the types `object` and `dynamic` are considered the same. Members declared in a single type cannot, therefore, differ in signature solely by `object` and `dynamic`.

The following example shows a set of overloaded method declarations along with their signatures.

```
interface ITest
    {
        void F();                       // F()
        void F(int x);                  // F(int)
        void F(ref int x);              // F(ref int)
        void F(out int x);              // F(out int)     error
```


www.it-ebooks.info

```
    void F(int x, int y);            // F(int, int)
    int F(string s);                 // F(string)
    int F(int x);                    // F(int)        error
    void F(string[] a);              // F(string[])
    void F(params string[] a);       // F(string[])   error
}
```

Note that any `ref` and `out` parameter modifiers (§10.6.1) are part of a signature. Thus `F(int)` and `F(ref int)` are unique signatures. However, `F(ref int)` and `F(out int)` cannot be declared within the same interface because their signatures differ solely by `ref` and `out`. Also, note that the return type and the `params` modifier are not part of a signature, so it is not possible to overload solely based on return type or on the inclusion or exclusion of the `params` modifier. As such, the declarations of the methods `F(int)` and `F(params string[])` identified above result in a compile-time error.

3.7 Scopes

The ***scope*** of a name is the region of program text within which it is possible to refer to the entity declared by the name without qualification of the name. Scopes can be ***nested***, and an inner scope may redeclare the meaning of a name from an outer scope (this does not, however, remove the restriction imposed by §3.3 that within a nested block it is not possible to declare a local variable with the same name as a local variable in an enclosing block). The name from the outer scope is then said to be ***hidden*** in the region of program text covered by the inner scope, and access to the outer name is possible only by qualifying the name.

- The scope of a namespace member declared by a *namespace-member-declaration* (§9.5) with no enclosing *namespace-declaration* is the entire program text.
- The scope of a namespace member declared by a *namespace-member-declaration* within a *namespace-declaration* whose fully qualified name is `N` is the *namespace-body* of every *namespace-declaration* whose fully qualified name is `N` or starts with `N`, followed by a period.
- The scope of a name defined by an *extern-alias-directive* extends over the *using-directives*, *global-attributes*, and *namespace-member-declarations* of its immediately containing compilation unit or namespace body. An *extern-alias-directive* does not contribute any new members to the underlying declaration space. In other words, an *extern-alias-directive* is not transitive, but rather affects only the compilation unit or namespace body in which it occurs.
- The scope of a name defined or imported by a *using-directive* (§9.4) extends over the *namespace-member-declarations* of the *compilation-unit* or *namespace-body* in which the

www.it-ebooks.info

using-directive occurs. A *using-directive* may make zero or more namespace or type names available within a particular *compilation-unit* or *namespace-body*, but does not contribute any new members to the underlying declaration space. In other words, a *using-directive* is not transitive, but rather affects only the *compilation-unit* or *namespace-body* in which it occurs.

- The scope of a type parameter declared by a *type-parameter-list* on a *class-declaration* (§10.1) is the *class-base*, *type-parameter-constraints-clauses*, and *class-body* of that *class-declaration*.
- The scope of a type parameter declared by a *type-parameter-list* on a *struct-declaration* (§11.1) is the *struct-interfaces*, *type-parameter-constraints-clauses*, and *struct-body* of that *struct-declaration*.
- The scope of a type parameter declared by a *type-parameter-list* on an *interface-declaration* (§13.1) is the *interface-base*, *type-parameter-constraints-clauses*, and *interface-body* of that *interface-declaration*.
- The scope of a type parameter declared by a *type-parameter-list* on a *delegate-declaration* (§15.1) is the *return-type*, *formal-parameter-list*, and *type-parameter-constraints-clauses* of that *delegate-declaration*.
- The scope of a member declared by a *class-member-declaration* (§10.1.6) is the *class-body* in which the declaration occurs. In addition, the scope of a class member extends to the *class-body* of those derived classes that are included in the accessibility domain (§3.5.2) of the member.
- The scope of a member declared by a *struct-member-declaration* (§11.2) is the *struct-body* in which the declaration occurs.
- The scope of a member declared by an *enum-member-declaration* (§14.3) is the *enum-body* in which the declaration occurs.
- The scope of a parameter declared in a *method-declaration* (§10.6) is the *method-body* of that *method-declaration*.
- The scope of a parameter declared in an *indexer-declaration* (§10.9) is the *accessor-declarations* of that *indexer-declaration*.
- The scope of a parameter declared in an *operator-declaration* (§10.10) is the *block* of that *operator-declaration*.
- The scope of a parameter declared in a *constructor-declaration* (§10.11) is the *constructor-initializer* and *block* of that *constructor-declaration*.
- The scope of a parameter declared in a *lambda-expression* (§7.15) is the *lambda-expression-body* of that *lambda-expression*.
- The scope of a parameter declared in an *anonymous-method-expression* (§7.15) is the *block* of that *anonymous-method-expression*.

www.it-ebooks.info

- The scope of a label declared in a *labeled-statement* (§8.4) is the *block* in which the declaration occurs.
- The scope of a local variable declared in a *local-variable-declaration* (§8.5.1) is the *block* in which the declaration occurs.
- The scope of a local variable declared in a *switch-block* of a `switch` statement (§8.7.2) is the *switch-block*.
- The scope of a local variable declared in a *for-initializer* of a `for` statement (§8.8.3) is the *for-initializer*, the *for-condition*, the *for-iterator*, and the contained *statement* of the `for` statement.
- The scope of a local constant declared in a *local-constant-declaration* (§8.5.2) is the *block* in which the declaration occurs. It is a compile-time error to refer to a local constant in a textual position that precedes its *constant-declarator*.
- The scope of a variable declared as part of a *foreach-statement*, *using-statement*, *lock-statement*, or *query-expression* is determined by the expansion of the given construct.

JESSE LIBERTY This list is a classic, almost iconic example of the difference between what is possible and what is advisable. Scope, like method names, and nearly everything else in your program should be as self-revealing and unambiguous as possible—but no more so!

Within the scope of a namespace, class, struct, or enumeration member, it is possible to refer to the member in a textual position that precedes the declaration of the member. For example, in

```
class A
{
    void F()
    {
        i = 1;
    }

    int i = 0;
}
```

it is valid for `F` to refer to `i` before it is declared.

Within the scope of a local variable, it is a compile-time error to refer to the local variable in a textual position that precedes the *local-variable-declarator* of the local variable. For example:

```
class A
{
    int i = 0;
```


www.it-ebooks.info

```
    void F()
    {
        i = 1;              // Error: use precedes declaration
        int i;
        i = 2;
    }

    void G()
    {
        int j = (j = 1);    // Valid
    }

    void H()
    {
        int a = 1, b = ++a; // Valid
    }
}
```

In the `F` method above, the first assignment to `i` specifically does not refer to the field declared in the outer scope. Rather, it refers to the local variable and results in a compile-time error because it textually precedes the declaration of the variable. In the `G` method, the use of `j` in the initializer for the declaration of `j` is valid because the use does not precede the *local-variable-declarator*. In the `H` method, a subsequent *local-variable-declarator* correctly refers to a local variable declared in an earlier *local-variable-declarator* within the same *local-variable-declaration*.

The scoping rules for local variables are designed to guarantee that the meaning of a name used in an expression context is always the same within a block. If the scope of a local variable were to extend only from its declaration to the end of the block, then in the example above, the first assignment would assign to the instance variable and the second assignment would assign to the local variable, possibly leading to compile-time errors if the statements of the block were later rearranged.

The meaning of a name within a block may differ based on the context in which the name is used. In the example

```
using System;

class A { }

class Test
{
    static void Main()
    {
        string A = "hello, world";
        string s = A;                   // Expression context

        Type t = typeof(A);             // Type context

        Console.WriteLine(s);           // Writes "hello, world"
        Console.WriteLine(t);           // Writes "A"
    }
}
```


www.it-ebooks.info

the name A is used in an expression context to refer to the local variable A and in a type context to refer to the class A.

3.7.1 Name Hiding

The scope of an entity typically encompasses more program text than the declaration space of the entity. In particular, the scope of an entity may include declarations that introduce new declaration spaces containing entities of the same name. Such declarations cause the original entity to become ***hidden***. Conversely, an entity is said to be ***visible*** when it is not hidden.

Name hiding occurs when scopes overlap through nesting and when scopes overlap through inheritance. The characteristics of the two types of hiding are described in the following sections.

> **JESSE LIBERTY** I would argue that name hiding should always be considered a bug, in that it makes for code that is miserably difficult to maintain and it is always avoidable.

3.7.1.1 *Hiding Through Nesting*

Name hiding through nesting can occur as a result of nesting namespaces or types within namespaces, as a result of nesting types within classes or structs, and as a result of parameter and local variable declarations.

In the example

```
class A
{
    int i = 0;

    void F()
    {
        int i = 1;
    }

    void G()
    {
        i = 1;
    }
}
```

within the F method, the instance variable i is hidden by the local variable i, but within the G method, i still refers to the instance variable.

When a name in an inner scope hides a name in an outer scope, it hides all overloaded occurrences of that name. In the example

www.it-ebooks.info

```
class Outer
{
    static void F(int i) { }

    static void F(string s) { }

    class Inner
    {
        void G()
        {
            F(1);                   // Invokes Outer.Inner.F
            F("Hello");             // Error
        }

        static void F(long l) { }
    }
}
```

the call `F(1)` invokes the `F` declared in `Inner` because all outer occurrences of `F` are hidden by the inner declaration. For the same reason, the call `F("Hello")` results in a compile-time error.

VLADIMIR RESHETNIKOV If a nested scope contains a member with the same name as a member from an outer scope, then the member from the outer scope is not always hidden due to the following rule: If the member is *invoked*, all *non-invocable* members are removed from the set (see §7.3).

```
class A
{
    static void Foo() { }
    class B
    {
        const int Foo = 1;
        void Bar()
        {
            Foo(); // Okay
        }
    }
}
```

3.7.1.2 *Hiding Through Inheritance*

Name hiding through inheritance occurs when classes or structs redeclare names that were inherited from base classes. This type of name hiding takes one of the following forms:

- A constant, field, property, event, or type introduced in a class or struct hides all base class members with the same name.

www.it-ebooks.info

- A method introduced in a class or struct hides all nonmethod base class members with the same name, and all base class methods with the same signature (method name and parameter count, modifiers, and types).
- An indexer introduced in a class or struct hides all base class indexers with the same signature (parameter count and types).

The rules governing operator declarations (§10.10) make it impossible for a derived class to declare an operator with the same signature as an operator in a base class. Thus operators never hide one another.

Contrary to hiding a name from an outer scope, hiding an accessible name from an inherited scope causes a warning to be reported. In the example

```
class Base
{
    public void F() { }
}

class Derived : Base
{
    public void F() { }             // Warning: hiding an inherited name
}
```

the declaration of `F` in `Derived` causes a warning to be reported. Hiding an inherited name is specifically not an error, since that would preclude separate evolution of base classes. For example, the above situation might have come about because a later version of `Base` introduced an `F` method that wasn't present in an earlier version of the class. Had the above situation been an error, then *any* change made to a base class in a separately versioned class library could potentially cause derived classes to become invalid.

The warning caused by hiding an inherited name can be eliminated through use of the `new` modifier:

```
class Base
{
    public void F() { }
}

class Derived : Base
{
    new public void F() { }
}
```

The `new` modifier indicates that the `F` in `Derived` is "new," and that it is indeed intended to hide the inherited member.

A declaration of a new member hides an inherited member only within the scope of the new member.

www.it-ebooks.info

```
class Base
{
    public static void F() { }
}

class Derived : Base
{
    new private static void F() { }          // Hides Base.F in Derived only
}

class MoreDerived : Derived
{
    static void G() { F(); }                 // Invokes Base.F
}
```

In the example above, the declaration of `F` in `Derived` hides the `F` that was inherited from `Base`, but since the new `F` in `Derived` has `private` access, its scope does not extend to `MoreDerived`. Thus the call `F()` in `MoreDerived.G` is valid and will invoke `Base.F`.

CHRIS SELLS If you find yourself using `new` to hide an instance method on the base class, you're almost always going to be disappointed, if for no other reason than a caller can simply cast to the base class to get to the "hidden" method. For example:

```
class Base { public void F() {} }
class Derived : Base { new public void F() {} }
Derived d = new Derived();
((Base)d).F();          // Base.F not so hidden as you would like
```

You'll be much happier if you pick a new name for the method in the derived class instead.

3.8 Namespace and Type Names

Several contexts in a C# program require a *namespace-name* or a *type-name* to be specified.

namespace-name:
    namespace-or-type-name

type-name:
    namespace-or-type-name

namespace-or-type-name:
    identifier type-argument-list$_{opt}$
    namespace-or-type-name . *identifier type-argument-list*$_{opt}$
    qualified-alias-member

www.it-ebooks.info

A *namespace-name* is a *namespace-or-type-name* that refers to a namespace. Following resolution as described below, the *namespace-or-type-name* of a *namespace-name* must refer to a namespace; otherwise, a compile-time error occurs. No type arguments (§4.4.1) can be present in a *namespace-name* (only types can have type arguments).

A *type-name* is a *namespace-or-type-name* that refers to a type. Following resolution as described below, the *namespace-or-type-name* of a *type-name* must refer to a type; otherwise, a compile-time error occurs.

If the *namespace-or-type-name* is a *qualified-alias-member*, its meaning is as described in §9.7. Otherwise, a *namespace-or-type-name* has one of four forms:

- `I`
- `I<`A_1`, ..., `A_k`>`
- `N.I`
- `N.I<`A_1`, ..., `A_k`>`

where `I` is a single identifier, `N` is a *namespace-or-type-name*, and `<`A_1`, ..., `A_k`>` is an optional *type-argument-list*. When no *type-argument-list* is specified, consider `K` to be zero.

The meaning of a *namespace-or-type-name* is determined as follows:

- If the *namespace-or-type-name* is of the form `I` or of the form `I<`A_1`, ..., `A_k`>`:
  - If `K` is zero and the *namespace-or-type-name* appears within a generic method declaration (§10.6) and if that declaration includes a type parameter (§10.1.3) with name `I`, then the *namespace-or-type-name* refers to that type parameter.
  - Otherwise, if the *namespace-or-type-name* appears within a type declaration, then for each instance type `T` (§10.3.1), starting with the instance type of that type declaration and continuing with the instance type of each enclosing class or struct declaration (if any):
    - If `K` is zero and the declaration of `T` includes a type parameter with name `I`, then the *namespace-or-type-name* refers to that type parameter.
    - Otherwise, if the *namespace-or-type-name* appears within the body of the type declaration, and `T` or any of its base types contain a nested accessible type having name `I` and `K` type parameters, then the *namespace-or-type-name* refers to that type constructed with the given type arguments. If there is more than one such type, the type declared within the more derived type is selected. Note that nontype members (constants, fields, methods, properties, indexers, operators, instance constructors, destructors, and static constructors) and type members with a different number of type parameters are ignored when determining the meaning of the *namespace-or-type-name*.

www.it-ebooks.info

- If the previous steps were unsuccessful, then, for each namespace `N`, starting with the namespace in which the *namespace-or-type-name* occurs, continuing with each enclosing namespace (if any), and ending with the global namespace, the following steps are evaluated until an entity is located:
  - If `K` is zero and `I` is the name of a namespace in `N`, then:
    - If the location where the *namespace-or-type-name* occurs is enclosed by a namespace declaration for `N` and the namespace declaration contains an *extern-alias-directive* or *using-alias-directive* that associates the name `I` with a namespace or type, then the *namespace-or-type-name* is ambiguous and a compile-time error occurs.
    - Otherwise, the *namespace-or-type-name* refers to the namespace named `I` in `N`.
  - Otherwise, if `N` contains an accessible type having name `I` and `K` type parameters, then:
    - If `K` is zero and the location where the *namespace-or-type-name* occurs is enclosed by a namespace declaration for `N` and the namespace declaration contains an *extern-alias-directive* or *using-alias-directive* that associates the name `I` with a namespace or type, then the *namespace-or-type-name* is ambiguous and a compile-time error occurs.
    - Otherwise, the *namespace-or-type-name* refers to the type constructed with the given type arguments.
  - Otherwise, if the location where the *namespace-or-type-name* occurs is enclosed by a namespace declaration for `N`:
    - If `K` is zero and the namespace declaration contains an *extern-alias-directive* or *using-alias-directive* that associates the name `I` with an imported namespace or type, then the *namespace-or-type-name* refers to that namespace or type.
    - Otherwise, if the namespaces imported by the *using-namespace-directive*s of the namespace declaration contain exactly one type having name `I` and `K` type parameters, then the *namespace-or-type-name* refers to that type constructed with the given type arguments.
    - Otherwise, if the namespaces imported by the *using-namespace-directive*s of the namespace declaration contain more than one type having name `I` and `K` type parameters, then the *namespace-or-type-name* is ambiguous and an error occurs.
- Otherwise, the *namespace-or-type-name* is undefined and a compile-time error occurs.

www.it-ebooks.info

- Otherwise, the *namespace-or-type-name* is of the form `N.I` or of the form `N.I<`A_1`, ..., `A_K`>`. `N` is first resolved as a *namespace-or-type-name*. If the resolution of `N` is not successful, a compile-time error occurs. Otherwise, `N.I` or `N.I<`A_1`, ..., `A_K`>` is resolved as follows:
  - If `K` is zero and `N` refers to a namespace and `N` contains a nested namespace with name `I`, then the *namespace-or-type-name* refers to that nested namespace.
  - Otherwise, if `N` refers to a namespace and `N` contains an accessible type having name `I` and `K` type parameters, then the *namespace-or-type-name* refers to that type constructed with the given type arguments.
  - Otherwise, if `N` refers to a (possibly constructed) class or struct type and `N` or any of its base classes contain a nested accessible type having name `I` and `K` type parameters, then the *namespace-or-type-name* refers to that type constructed with the given type arguments. If there is more than one such type, the type declared within the more derived type is selected. Note that if the meaning of `N.I` is being determined as part of resolving the base class specification of `N`, then the direct base class of `N` is considered to be object (§10.1.4.1).
  - Otherwise, `N.I` is an invalid *namespace-or-type-name*, and a compile-time error occurs.

VLADIMIR RESHETNIKOV This algorithm is different from (albeit similar to) the corresponding algorithm for simple names (see §7.6.2). It means that in contrived cases, the same identifier may have different meanings in contexts of *type-name* and *simple-name*:

```
class T
{
    public const int X = 1;
}
class C
{
    void Foo<T>(int x = T.X /* global::T */,
            T y = default(T) /* type parameter */) { }
}
```

A *namespace-or-type-name* is permitted to reference a static class (§10.1.1.3) only if

- The *namespace-or-type-name* is the `T` in a *namespace-or-type-name* of the form `T.I`, or
- The *namespace-or-type-name* is the `T` in a *typeof-expression* (§7.5.11) of the form `typeof(T)`.

www.it-ebooks.info

3.8.1 Fully Qualified Names

Every namespace and type has a ***fully qualified name***, which uniquely identifies the namespace or type among all others. The fully qualified name of a namespace or type `N` is determined as follows:

- If `N` is a member of the global namespace, its fully qualified name is `N`.
- Otherwise, its fully qualified name is `S.N`, where `S` is the fully qualified name of the namespace or type in which `N` is declared.

In other words, the fully qualified name of `N` is the complete hierarchical path of identifiers that lead to `N`, starting from the global namespace. Because every member of a namespace or type must have a unique name, it follows that the fully qualified name of a namespace or type is always unique.

The example below shows several namespace and type declarations, along with their associated fully qualified names.

```
class A { }                 // A

namespace X                 // X
 {
     class B                // X.B
     {
         class C { }        // X.B.C
     }

     namespace Y            // X.Y
     {
         class D { }        // X.Y.D
     }
 }

 namespace X.Y              // X.Y
 {
     class E { }            // X.Y.E
 }
```

JOSEPH ALBAHARI If a fully qualified name conflicts with a partially qualified or unqualified name (a nested accessible type, for instance), the latter wins. Prefixing the name with `global::` forces the fully qualified name to win (§9.7). There is little chance of such a collision in human-written code; with machine-written code, however, the odds are greater. For this reason, some code generators in design tools and IDEs emit the `global::` prefix before all fully qualified type names to eliminate any possibility of conflict.

www.it-ebooks.info

3.9 Automatic Memory Management

C# employs automatic memory management, which frees developers from manually allocating and freeing the memory occupied by objects. Automatic memory management policies are implemented by a ***garbage collector***. The memory management life cycle of an object is as follows:

1. When the object is created, memory is allocated for it, the constructor is run, and the object is considered live.
2. If the object, or any part of it, cannot be accessed by any possible continuation of execution, other than the running of destructors, the object is considered no longer in use, and it becomes eligible for destruction. The C# compiler and the garbage collector may choose to analyze code to determine which references to an object may be used in the future. For instance, if a local variable that is in scope is the only existing reference to an object, but that local variable is never referred to in any possible continuation of execution from the current execution point in the procedure, the garbage collector may (but is not required to) treat the object as no longer in use.

> **CHRISTIAN NAGEL** From a destructor, the C# compiler creates code to override the `Finalize` method of the base class. Overriding the `Finalize` method also means an overhead on object instantiation and keeps the object longer alive until the finalization was run. With C++/CLI, the code generated from the destructor implements the `IDisposable` interface, which has a better fit with deterministic cleanup.

> **JON SKEET** The point at which an object is eligible for destruction is earlier than you might expect. In particular, an object's destructor may run while another thread is still executing an instance method "in" the same object—so long as that instance method doesn't refer to any instance variables in any possible code path after the current point of execution.
>
> Fortunately, this behavior can cause a noticeable problem only in types with destructors, which are relatively uncommon in .NET code since the advent of `SafeHandle`.

3. Once the object is eligible for destruction, at some unspecified later time the destructor (§10.13) (if any) for the object is run. Unless overridden by explicit calls, the destructor for the object is run once only.

www.it-ebooks.info

CHRISTIAN NAGEL "The destruction is called at some unspecified later time . . ." When a destructor is created, it's usually a good idea to implement the interface `IDisposable` as well. With the `IDisposable` interface, the caller has a chance to invoke the cleanup code early, when the object is no longer needed.

CHRISTIAN NAGEL To invoke the destructor more than once, `GC.ReRegisterForFinalize` keeps the object alive and allows for the destructor to be called once more. Instead of taking this tack, it could be worthwhile to change the application architecture.

4. Once the destructor for an object is run, if that object, or any part of it, cannot be accessed by any possible continuation of execution, including the running of destructors, the object is considered inaccessible and the object becomes eligible for collection.
5. Finally, at some time after the object becomes eligible for collection, the garbage collector frees the memory associated with that object.

The garbage collector maintains information about object usage, and uses this information to make memory management decisions, such as where in memory to locate a newly created object, when to relocate an object, and when an object is no longer in use or inaccessible.

Like other languages that assume the existence of a garbage collector, C# is designed so that the garbage collector may implement a wide range of memory management policies. For instance, C# does not require that destructors be run or that objects be collected as soon as they are eligible, or that destructors be run in any particular order, or on any particular thread.

The behavior of the garbage collector can be controlled, to some degree, via static methods on the class `System.GC`. This class can be used to request a collection to occur, destructors to be run (or not run), and so forth.

ERIC LIPPERT Using these static methods to control the behavior of the garbage collector is almost never a good idea. In production code, odds are good that the garbage collector knows more about when would be a good time to do a collection than your program does.

Explicit tweaking of the garbage collector behavior at runtime should typically be limited to purposes such as forcing a collection for testing purposes.

www.it-ebooks.info

Since the garbage collector is allowed wide latitude in deciding when to collect objects and run destructors, a conforming implementation may produce output that differs from that shown by the following code. The program

```
using System;

class A
{
    ~A()
    {
        Console.WriteLine("Destruct instance of A");
    }
}

class B
{
    object Ref;

    public B(object o)
    {
        Ref = o;
    }

    ~B()
    {
        Console.WriteLine("Destruct instance of B");
    }
}

class Test
{
    static void Main()
    {
        B b = new B(new A());
        b = null;
        GC.Collect();
        GC.WaitForPendingFinalizers();
    }
}
```

creates an instance of class `A` and an instance of class `B`. These objects become eligible for garbage collection when the variable `b` is assigned the value `null`, since after this time it is impossible for any user-written code to access them. The output could be either

```
Destruct instance of A
Destruct instance of B
```

or

```
Destruct instance of B
Destruct instance of A
```

because the language imposes no constraints on the order in which objects are garbage collected.

www.it-ebooks.info

In subtle cases, the distinction between "eligible for destruction" and "eligible for collection" can be important. For example:

```
using System;

using System;
class A
{
    ~A() {
        Console.WriteLine("Destruct instance of A");
    }

    public void F() {
        Console.WriteLine("A.F");
        Test.RefA = this;
    }
}

class B
{
    public A Ref;

    ~B() {
        Console.WriteLine("Destruct instance of B");
        Ref.F();
    }
}

class Test
{
    public static A RefA;
    public static B RefB;

    static void Main() {
        RefB = new B();
        RefA = new A();
        RefB.Ref = RefA;
        RefB = null;
        RefA = null;

        // A and B now eligible for destruction
        GC.Collect();
        GC.WaitForPendingFinalizers();

        // B now eligible for collection, but A is not
        if (RefA != null)
            Console.WriteLine("RefA is not null");
}
```

In the above program, if the garbage collector chooses to run the destructor of `A` before the destructor of `B`, then the output of this program might be

```
Destruct instance of A
Destruct instance of B
A.F
RefA is not null
```


www.it-ebooks.info

Note that although the instance of `A` was not in use and `A`'s destructor was run, it is still possible for methods of `A` (in this case, `F`) to be called from another destructor. Also, note that running a destructor may cause an object to become usable from the mainline program again. In this case, the running of `B`'s destructor caused an instance of `A` that was previously not in use to become accessible from the live reference `Test.RefA`. After the call to `WaitForPendingFinalizers`, the instance of `B` is eligible for collection, but the instance of `A` is not, because of the reference `Test.RefA`.

To avoid confusion and unexpected behavior, it is generally a good idea for destructors to only perform cleanup on data stored in their object's own fields, and not to perform any actions on referenced objects or static fields.

ERIC LIPPERT It is an even better idea for the destructor to clean up only the fields that contain data representing unmanaged objects, such as operating system handles. Because you do not know which thread the destructor will run on or when it will run, it is particularly important that the destructor have as few side effects as possible.

The calls to `Console.WriteLine` in the example obviously violate this good advice to only do cleanup and not perform other actions. This code is intended solely as a pedagogic aid. Real production code destructors should never attempt to do anything that has a complex side effect such as console output.

An alternative to using destructors is to let a class implement the `System.IDisposable` interface. This allows the client of the object to determine when to release the resources of the object, typically by accessing the object as a resource in a `using` statement (§8.13).

BRAD ABRAMS Nine times out of ten, using `GC.Collect()` is a mistake. It is often an indication of a poor design that is being cobbled together. The garbage collector is a finely tuned instrument, like a Porsche. Just as you would not play bumper tag with a new Porsche, so you should generally avoid interfering with the garbage collector's algorithms. The garbage collector is designed to unobtrusively step in at the right time and collect the most important unused memory. Kicking it with the call `GC.Collect()` can throw off the balance and tuning. Before resorting to this solution, take a few minutes to figure why it is required. Have you disposed of all your instances? Have you dropped references where you could? Have you used weak references in the right places?

www.it-ebooks.info

KRZYSZTOF CWALINA Some people attribute the performance problems of modern VM-based systems to their use of garbage collection. In fact, modern garbage collectors are so efficient that there is very little software left that would have problems with the garbage collector's performance in itself. The biggest performance culprit is over-engineering of applications and, sadly, many framework libraries.

CHRIS SELLS In the modern business systems that .NET is generally used to build, garbage collection provides wondrous robustness. It's only now that we have begun to use managed code in real-time applications, such as games (e.g., XNA for Xbox) and the Windows Phone 7 platforms, that the careful consideration of the behavior of the garbage collector has become important. Even in those cases, an efficient design is much more about pre-allocating resources before the twitch begins than with monkeying directly with the garbage collector.

CHRISTIAN NAGEL To reemphasize the point made by Krzysztof, discussions on the performance of the garbage collector usually are similar to discussions some years ago when C code was compared to assembly code, or some years later when C++ performance was suspect in comparison to C. Of course, there is still a place for assembler code and for C, but most applications are happy with a managed runtime.

BILL WAGNER Collectively, these notes point out how few of your regular assumptions are valid in the context of a destructor. Member variables may have already executed their destructors. They are called on a different thread, so thread local storage may not be valid. They are called by the system, so your application won't see errors reported by destructors using exceptions. It's hard to over-emphasize how defensively you need to write destructors. Luckily, they are needed only rarely.

3.10 Execution Order

Execution of a C# program proceeds such that the side effects of each executing thread are preserved at critical execution points. A ***side effect*** is defined as a read or write of a volatile field, a write to a nonvolatile variable, a write to an external resource, and the throwing of an exception. The critical execution points at which the order of these side effects must be preserved are references to volatile fields (§10.5.3), `lock` statements (§8.12), and thread

www.it-ebooks.info

creation and termination. The execution environment is free to change the order of execution of a C# program, subject to the following constraints:

- Data dependence is preserved within a thread of execution. That is, the value of each variable is computed as if all statements in the thread were executed in original program order.
- Initialization ordering rules are preserved (§10.5.4 and §10.5.5).
- The ordering of side effects is preserved with respect to volatile reads and writes (§10.5.3). Additionally, the execution environment need not evaluate part of an expression if it can deduce that that expression's value is not used and that no needed side effects are produced (including any caused by calling a method or accessing a volatile field). When program execution is interrupted by an asynchronous event (such as an exception thrown by another thread), it is not guaranteed that the observable side effects will be visible in the original program order.

www.it-ebooks.info

4. Types

The types of the C# language are divided into two main categories: ***value types*** and ***reference types***. Both value types and reference types may be ***generic types***, which take one or more ***type parameters***. Type parameters can designate both value types and reference types.

type:
    value-type
    reference-type
    type-parameter

A third category of types, pointers, is available only in unsafe code. This issue is discussed further in §18.2.

Value types differ from reference types in that variables of the value types directly contain their data, whereas variables of the reference types store ***references*** to their data, the latter being known as ***objects***. With reference types, it is possible for two variables to reference the same object, and thus possible for operations on one variable to affect the object referenced by the other variable. With value types, the variables each have their own copy of the data, so it is not possible for operations on one to affect the other.

C#'s type system is unified such that *a value of any type can be treated as an object*. Every type in C# directly or indirectly derives from the `object` class type, and `object` is the ultimate base class of all types. Values of reference types are treated as objects simply by viewing the values as type `object`. Values of value types are treated as objects by performing boxing and unboxing operations (§4.3).

ERIC LIPPERT We normally do not think of interface types or the types associated with type parameters as having a "base class" per se. What this discussion is getting at is that every concrete object—no matter how you are treating it at compile time—may be treated as an instance of `object` at runtime.

www.it-ebooks.info

4.1 Value Types

A value type is either a struct type or an enumeration type. C# provides a set of predefined struct types called the ***simple types***. The simple types are identified through reserved words.

value-type:
    struct-type
    enum-type

struct-type:
    type-name
    simple-type
    nullable-type

simple-type:
    numeric-type
    `bool`

numeric-type:
    integral-type
    floating-point-type
    `decimal`

integral-type:
    `sbyte`
    `byte`
    `short`
    `ushort`
    `int`
    `uint`
    `long`
    `ulong`
    `char`

floating-point-type:
    `float`
    `double`

nullable-type:
    non-nullable-value-type `?`

non-nullable-value-type:
    type

enum-type:
    type-name

www.it-ebooks.info

Unlike a variable of a reference type, a variable of a value type can contain the value `null` only if the value type is a nullable type. For every non-nullable value type, there is a corresponding nullable value type denoting the same set of values plus the value `null`.

Assignment to a variable of a value type creates a *copy* of the value being assigned. This differs from assignment to a variable of a reference type, which copies the reference but not the object identified by the reference.

4.1.1 The System.ValueType Type

All value types implicitly inherit from the class `System.ValueType`, which in turn inherits from class `object`. It is not possible for any type to derive from a value type, and value types are thus implicitly sealed (§10.1.1.2).

Note that `System.ValueType` is not itself a *value-type*. Rather, it is a *class-type* from which all *value-type*s are automatically derived.

> **ERIC LIPPERT** This point is frequently confusing to novices. I am often asked, "But how is it possible that a value type derives from a reference type?" I think the confusion arises as a result of a misunderstanding of what "derives from" means. Derivation does not imply that the layout of the bits in memory of the base type is somewhere found in the layout of bits in the derived type. Rather, it simply implies that some mechanism exists whereby members of the base type may be accessed from the derived type.

4.1.2 Default Constructors

All value types implicitly declare a public parameterless instance constructor called the ***default constructor***. The default constructor returns a zero-initialized instance known as the ***default value*** for the value type:

- For all *simple-types*, the default value is the value produced by a bit pattern of all zeros:
  - For `sbyte`, `byte`, `short`, `ushort`, `int`, `uint`, `long`, and `ulong`, the default value is `0`.
  - For `char`, the default value is `'\x0000'`.
  - For `float`, the default value is `0.0f`.
  - For `double`, the default value is `0.0d`.
  - For `decimal`, the default value is `0.0m`.
  - For `bool`, the default value is `false`.

www.it-ebooks.info

- For an *enum-type* `E`, the default value is `0`, converted to the type `E`.
- For a *struct-type*, the default value is the value produced by setting all value type fields to their default values and all reference type fields to `null`.

> **VLADIMIR RESHETNIKOV** Obviously, the wording "all fields" here means only instance fields (not static fields). It also includes field-like instance events, if any exist.

- For a *nullable-type*, the default value is an instance for which the `HasValue` property is false and the `Value` property is undefined. The default value is also known as the ***null value*** of the nullable type.

Like any other instance constructor, the default constructor of a value type is invoked using the `new` operator. For efficiency reasons, this requirement is not intended to actually have the implementation generate a constructor call. In the example below, variables `i` and `j` are both initialized to zero.

```
class A
{
    void F() {
        int i = 0;
        int j = new int();
    }
}
```

Because every value type implicitly has a public parameterless instance constructor, it is not possible for a struct type to contain an explicit declaration of a parameterless constructor. A struct type is, however, permitted to declare parameterized instance constructors (§11.3.8).

> **ERIC LIPPERT** Another good way to obtain the default value of a type is to use the `default(type)` expression.

> **JON SKEET** This is one example of where the C# language and the underlying platform may have different ideas. If you ask the .NET platform for the constructors of a value type, you usually won't find a parameterless one. Instead, .NET has a specific instruction for initializing the default value for a value type. Usually these small impedence mismatches have no effect on developers, but it's good to know that they're possible—and that they don't represent a fault in either specification.

www.it-ebooks.info

4.1.3 Struct Types

A struct type is a value type that can declare constants, fields, methods, properties, indexers, operators, instance constructors, static constructors, and nested types. The declaration of struct types is described in §11.1.

4.1.4 Simple Types

C# provides a set of predefined struct types called the ***simple types***. The simple types are identified through reserved words, but these reserved words are simply aliases for predefined struct types in the `System` namespace, as described in the table below.

Reserved Word	Aliased Type
`sbyte`	`System.SByte`
`byte`	`System.Byte`
`short`	`System.Int16`
`ushort`	`System.UInt16`
`int`	`System.Int32`
`uint`	`System.UInt32`
`long`	`System.Int64`
`ulong`	`System.UInt64`
`char`	`System.Char`
`float`	`System.Single`
`double`	`System.Double`
`bool`	`System.Boolean`
`decimal`	`System.Decimal`

Because a simple type aliases a struct type, every simple type has members. For example, `int` has the members declared in `System.Int32` and the members inherited from `System.Object`, and the following statements are permitted:

```
int i = int.MaxValue;         // System.Int32.MaxValue constant
string s = i.ToString();      // System.Int32.ToString() instance method
string t = 123.ToString();    // System.Int32.ToString() instance method
```


www.it-ebooks.info

The simple types differ from other struct types in that they permit certain additional operations:

- Most simple types permit values to be created by writing *literals* (§2.4.4). For example, `123` is a literal of type `int` and `'a'` is a literal of type `char`. C# makes no provision for literals of struct types in general, and nondefault values of other struct types are ultimately always created through instance constructors of those struct types.

> **ERIC LIPPERT** The "most" in the phrase "most simple types" refers to the fact that some simple types, such as `short`, have no literal form. In reality, any integer literal small enough to fit into a `short` is implicitly converted to a `short` when used as one, so in that sense there are literal values for all simple types.
>
> There are a handful of possible values for simple types that have no literal forms. The NaN (Not-a-Number) values for floating point types, for example, have no literal form.

- When the operands of an expression are all simple type constants, it is possible for the compiler to evaluate the expression at compile time. Such an expression is known as a *constant-expression* (§7.19). Expressions involving operators defined by other struct types are not considered to be constant expressions.

> **VLADIMIR RESHETNIKOV** It is not just "possible": The compiler always **does** fully evaluate *constant-expressions* at compile time.

- Through `const` declarations, it is possible to declare constants of the simple types (§10.4). It is not possible to have constants of other struct types, but a similar effect is provided by `static readonly` fields.
- Conversions involving simple types can participate in evaluation of conversion operators defined by other struct types, but a user-defined conversion operator can never participate in evaluation of another user-defined operator (§6.4.3).

> **JOSEPH ALBAHARI** The simple types also provide a means by which the compiler can leverage direct support within the IL (and ultimately the processor) for computations on integer and floating point values. This scheme allows arithmetic on simple types that have processor support (typically `float`, `double`, and the integral types) to run at native speed.

www.it-ebooks.info

4.1.5 Integral Types

C# supports nine integral types: `sbyte`, `byte`, `short`, `ushort`, `int`, `uint`, `long`, `ulong`, and `char`. The integral types have the following sizes and ranges of values:

- The `sbyte` type represents signed 8-bit integers with values between –128 and 127.
- The `byte` type represents unsigned 8-bit integers with values between 0 and 255.
- The `short` type represents signed 16-bit integers with values between –32768 and 32767.
- The `ushort` type represents unsigned 16-bit integers with values between 0 and 65535.
- The `int` type represents signed 32-bit integers with values between –2147483648 and 2147483647.
- The `uint` type represents unsigned 32-bit integers with values between 0 and 4294967295.
- The `long` type represents signed 64-bit integers with values between –9223372036854775808 and 9223372036854775807.
- The `ulong` type represents unsigned 64-bit integers with values between 0 and 18446744073709551615.
- The `char` type represents unsigned 16-bit integers with values between 0 and 65535. The set of possible values for the `char` type corresponds to the Unicode character set. Although `char` has the same representation as `ushort`, not all operations permitted on one type are permitted on the other.

> **JESSE LIBERTY** I have to confess that with the power of modern PCs, and the greater cost of programmer time relative to the cost of memory, I tend to use `int` for just about any integral (nonfractional) value and `double` for any fractional value. All the rest, I pretty much ignore.

The integral-type unary and binary operators always operate with signed 32-bit precision, unsigned 32-bit precision, signed 64-bit precision, or unsigned 64-bit precision:

- For the unary `+` and `~` operators, the operand is converted to type `T`, where `T` is the first of `int`, `uint`, `long`, and `ulong` that can fully represent all possible values of the operand. The operation is then performed using the precision of type `T`, and the type of the result is `T`.
- For the unary `-` operator, the operand is converted to type `T`, where `T` is the first of `int` and `long` that can fully represent all possible values of the operand. The operation is then

www.it-ebooks.info

performed using the precision of type `T`, and the type of the result is `T`. The unary `-` operator cannot be applied to operands of type `ulong`.

- For the binary `+`, `-`, `*`, `/`, `%`, `&`, `^`, `|`, `==`, `!=`, `>`, `<`, `>=`, and `<=` operators, the operands are converted to type `T`, where `T` is the first of `int`, `uint`, `long`, and `ulong` that can fully represent all possible values of both operands. The operation is then performed using the precision of type `T`, and the type of the result is `T` (or `bool` for the relational operators). It is not permitted for one operand to be of type `long` and the other to be of type `ulong` with the binary operators.
- For the binary `<<` and `>>` operators, the left operand is converted to type `T`, where `T` is the first of `int`, `uint`, `long`, and `ulong` that can fully represent all possible values of the operand. The operation is then performed using the precision of type `T`, and the type of the result is `T`.

The `char` type is classified as an integral type, but it differs from the other integral types in two ways:

- There are no implicit conversions from other types to the `char` type. In particular, even though the `sbyte`, `byte`, and `ushort` types have ranges of values that are fully representable using the `char` type, implicit conversions from `sbyte`, `byte`, or `ushort` to `char` do not exist.
- Constants of the `char` type must be written as *character-literal*s or as *integer-literal*s in combination with a cast to type `char`. For example, `(char)10` is the same as `'\x000A'`.

The `checked` and `unchecked` operators and statements are used to control overflow checking for integral-type arithmetic operations and conversions (§7.6.12). In a `checked` context, an overflow produces a compile-time error or causes a `System.OverflowException` to be thrown. In an `unchecked` context, overflows are ignored and any high-order bits that do not fit in the destination type are discarded.

4.1.6 Floating Point Types

C# supports two floating point types: `float` and `double`. The `float` and `double` types are represented using the 32-bit single-precision and 64-bit double-precision IEEE 754 formats, which provide the following sets of values:

- Positive zero and negative zero. In most situations, positive zero and negative zero behave identically as the simple value zero, but certain operations distinguish between the two (§7.8.2).

www.it-ebooks.info

VLADIMIR RESHETNIKOV Be aware that the default implementation of the `Equals` method in value types can use bitwise comparison in some cases to speed up performance. If two instances of your value type contain in their fields positive and negative zero, respectively, they can compare as not equal. You can override the `Equals` method to change the default behavior.

```
using System;

struct S
{
    double X;

    static void Main()
    {
        var a = new S {X = 0.0};
        var b = new S {X = -0.0};
        Console.WriteLine(a.X.Equals(b.X)); // True
        Console.WriteLine(a.Equals(b)); // False
    }
}
```

PETER SESTOFT Some of the confusion over negative zero may stem from the fact that the current implementations of C# print positive and negative zero in the same way, as `0.0`, and no combination of formatting parameters seems to affect that display. Although this is probably done with the best of intentions, it is unfortunate. To reveal a negative zero, you must resort to strange-looking code like this, which works because `1/(-0.0) = -Infinity < 0`:

```
public static string DoubleToString(double d) {
  if (d == 0.0 && 1/d < 0)
    return "-0.0";
  else
    return d.ToString();
}
```

- Positive infinity and negative infinity. Infinities are produced by such operations as dividing a non-zero number by zero. For example, `1.0 / 0.0` yields positive infinity, and `-1.0 / 0.0` yields negative infinity.
- The ***Not-a-Number*** value, often abbreviated NaN. NaNs are produced by invalid floating point operations, such as dividing zero by zero.

PETER SESTOFT A large number of distinct NaNs exist, each of which has a different "payload." See the annotations on §7.8.1.

www.it-ebooks.info

- The finite set of non-zero values of the form $s \times m \times 2^e$, where s is 1 or −1, and m and e are determined by the particular floating point type: For `float`, $0 < m < 2^{24}$ and $-149 \leq e \leq 104$; for `double`, $0 < m < 2^{53}$ and $-1075 \leq e \leq 970$. Denormalized floating point numbers are considered valid non-zero values.

The `float` type can represent values ranging from approximately 1.5×10^{-45} to 3.4×10^{38} with a precision of 7 digits.

The `double` type can represent values ranging from approximately 5.0×10^{-324} to 1.7×10^{308} with a precision of 15 or 16 digits.

If one of the operands of a binary operator is of a floating point type, then the other operand must be of an integral type or a floating point type, and the operation is evaluated as follows:

- If one of the operands is of an integral type, then that operand is converted to the floating point type of the other operand.
- Then, if either of the operands is of type `double`, the other operand is converted to `double`, the operation is performed using at least `double` range and precision, and the type of the result is `double` (or `bool` for the relational operators).
- Otherwise, the operation is performed using at least `float` range and precision, and the type of the result is `float` (or `bool` for the relational operators).

The floating point operators, including the assignment operators, never produce exceptions. Instead, in exceptional situations, floating point operations produce zero, infinity, or NaN, as described below:

- If the result of a floating point operation is too small for the destination format, the result of the operation becomes positive zero or negative zero.
- If the result of a floating point operation is too large for the destination format, the result of the operation becomes positive infinity or negative infinity.
- If a floating point operation is invalid, the result of the operation becomes NaN.
- If one or both operands of a floating point operation is NaN, the result of the operation becomes NaN.

Floating point operations may be performed with higher precision than the result type of the operation. For example, some hardware architectures support an "extended" or "long double" floating point type with greater range and precision than the `double` type, and implicitly perform all floating point operations using this higher precision type. Only at excessive cost in performance can such hardware architectures be made to perform floating point operations with *less* precision. Rather than require an implementation to forfeit

www.it-ebooks.info

both performance and precision, C# allows a higher precision type to be used for all floating point operations. Other than delivering more precise results, this rarely has any measurable effects. However, in expressions of the form `x * y / z`, where the multiplication produces a result that is outside the `double` range, but the subsequent division brings the temporary result back into the `double` range, the fact that the expression is evaluated in a higher range format may cause a finite result to be produced instead of an infinity.

JOSEPH ALBAHARI NaNs are sometimes used to represent special values. In Microsoft's Windows Presentation Foundation, `double.NaN` represents a measurement whose value is "automatic." Another way to represent such a value is with a nullable type; yet another is with a custom struct that wraps a numeric type and adds another field.

4.1.7 The decimal Type

The `decimal` type is a 128-bit data type suitable for financial and monetary calculations. The `decimal` type can represent values ranging from 1.0×10^{-28} to approximately 7.9×10^{28} with 28 or 29 significant digits.

The finite set of values of type `decimal` are of the form $(-1)^s \times c \times 10^{-e}$, where the sign s is 0 or 1, the coefficient c is given by $0 \le c < 2^{96}$, and the scale e is such that $0 \le e \le 28$.The `decimal` type does not support signed zeros, infinities, or NaNs. A `decimal` is represented as a 96-bit integer scaled by a power of 10. For `decimal`s with an absolute value less than `1.0m`, the value is exact to the 28th decimal place, but no further. For `decimal`s with an absolute value greater than or equal to `1.0m`, the value is exact to 28 or 29 digits. Unlike with the `float` and `double` data types, decimal fractional numbers such as 0.1 can be represented exactly in the `decimal` representation. In the `float` and `double` representations, such numbers are often infinite fractions, making those representations more prone to round-off errors.

PETER SESTOFT The IEEE 754-2008 standard describes a decimal floating point type called `decimal128`. It is similar to the type `decimal` described here, but packs a lot more punch within the same 128 bits. It has 34 significant decimal digits, a range from 10^{-6134} to 10^{6144}, and supports NaNs. It was designed by Mike Cowlishaw at IBM UK. Since it extends the current `decimal` in all respects, it would seem feasible for C# to switch to IEEE `decimal128` in some future version.

If one of the operands of a binary operator is of type `decimal`, then the other operand must be of an integral type or of type `decimal`. If an integral type operand is present, it is converted to `decimal` before the operation is performed.

www.it-ebooks.info

> **BILL WAGNER** You cannot mix `decimal` and the floating point types (`float`, `double`). This rule exists because you would lose precision mixing computations between those types. You must apply an explicit conversion when mixing `decimal` and floating point types.

The result of an operation on values of type `decimal` is what would result from calculating an exact result (preserving scale, as defined for each operator) and then rounding to fit the representation. Results are rounded to the nearest representable value and, when a result is equally close to two representable values, to the value that has an even number in the least significant digit position (this is known as "banker's rounding"). A zero result always has a sign of 0 and a scale of 0.

> **ERIC LIPPERT** This method has the attractive property that it typically introduces less bias than methods that always round down or up when there is a "tie" between two possibilities.
>
> Oddly enough, despite the nickname, there is little evidence that this method of rounding was ever in widespread use in banking.

If a `decimal` arithmetic operation produces a value less than or equal to 5×10^{-29} in absolute value, the result of the operation becomes zero. If a `decimal` arithmetic operation produces a result that is too large for the `decimal` format, a `System.OverflowException` is thrown.

The `decimal` type has greater precision but smaller range than the floating point types. Thus conversions from the floating point types to `decimal` might produce overflow exceptions, and conversions from `decimal` to the floating point types might cause loss of precision. For these reasons, no implicit conversions exist between the floating point types and `decimal`, and without explicit casts, it is not possible to mix floating point and `decimal` operands in the same expression.

> **ERIC LIPPERT** C# does not support the `Currency` data type familiar to users of Visual Basic 6 and other OLE Automation-based programming languages. Because `decimal` has both more range and precision than `Currency`, anything that you could have done with a `Currency` can be done just as well with a `decimal`.

4.1.8 The bool Type

The `bool` type represents boolean logical quantities. The possible values of type `bool` are `true` and `false`.

www.it-ebooks.info

No standard conversions exist between `bool` and other types. In particular, the `bool` type is distinct and separate from the integral types; a `bool` value cannot be used in place of an integral value, and vice versa.

In the C and C++ languages, a zero integral or floating point value, or a null pointer, can be converted to the boolean value `false`, and a non-zero integral or floating point value, or a non-null pointer, can be converted to the boolean value `true`. In C#, such conversions are accomplished by explicitly comparing an integral or floating point value to zero, or by explicitly comparing an object reference to `null`.

CHRIS SELLS The inability of a non-`bool` to be converted to a `bool` most often bites me when comparing for `null`. For example:

```
object obj = null;
if( obj ) { ... }            // Okay in C/C++, error in C#
if( obj != null ) { ... }    // Okay in C/C++/C#
```

4.1.9 Enumeration Types

An enumeration type is a distinct type with named constants. Every enumeration type has an underlying type, which must be `byte`, `sbyte`, `short`, `ushort`, `int`, `uint`, `long`, or `ulong`. The set of values of the enumeration type is the same as the set of values of the underlying type. Values of the enumeration type are not restricted to the values of the named constants. Enumeration types are defined through enumeration declarations (§14.1).

ERIC LIPPERT This is an important point: Nothing stops you from putting a value that is not in the enumerated type into a variable of that type. Do not rely on the language or the runtime environment to verify that instances of enumerated types are within the bounds you expect.

VLADIMIR RESHETNIKOV The CLR also supports `char` as an underlying type of an enumeration. If you happen to reference an assembly containing such a type in your application, the C# compiler will not recognize this type as an enumeration and will not allow you, for example, to convert it to or from an integral type.

4.1.10 Nullable Types

A nullable type can represent all values of its ***underlying type*** plus an additional null value. A nullable type is written `T?`, where `T` is the underlying type. This syntax is shorthand for `System.Nullable<T>`, and the two forms can be used interchangeably.

www.it-ebooks.info

A ***non-nullable value type***, conversely, is any value type other than `System.Nullable<T>` and its shorthand `T?` (for any `T`), plus any type parameter that is constrained to be a non-nullable value type (that is, any type parameter with a `struct` constraint). The `System.Nullable<T>` type specifies the value type constraint for `T` (§10.1.5), which means that the underlying type of a nullable type can be any non-nullable value type. The underlying type of a nullable type cannot be a nullable type or a reference type. For example, `int??` and `string?` are invalid types.

An instance of a nullable type `T?` has two public read-only properties:

- A `HasValue` property of type `bool`
- A `Value` property of type `T`

An instance for which `HasValue` is `true` is said to be non-null. A non-null instance contains a known value and `Value` returns that value.

An instance for which `HasValue` is `false` is said to be null. A null instance has an undefined value. Attempting to read the `Value` of a null instance causes a `System.InvalidOperationException` to be thrown. The process of accessing the `Value` property of a nullable instance is referred to as ***unwrapping***.

In addition to the default constructor, every nullable type `T?` has a public constructor that takes a single argument of type `T`. Given a value `x` of type `T`, a constructor invocation of the form

```
new T?(x)
```

creates a non-null instance of `T?` for which the `Value` property is `x`. The process of creating a non-null instance of a nullable type for a given value is referred to as ***wrapping***.

Implicit conversions are available from the `null` literal to `T?` (§6.1.5) and from `T` to `T?` (§6.1.4).

4.2 Reference Types

A reference type is a class type, an interface type, an array type, or a delegate type.

reference-type:
    class-type
    interface-type
    array-type
    delegate-type

www.it-ebooks.info

class-type:
    type-name
    `object`
    `dynamic`
    `string`

interface-type:
    type-name

array-type:
    non-array-type rank-specifiers

non-array-type:
    type

rank-specifiers:
    rank-specifier
    rank-specifiers rank-specifier

rank-specifier:
    `[` *dim-separators*$_{opt}$ `]`

dim-separators:
    `,`
    dim-separators `,`

delegate-type:
    type-name

A reference type value is a reference to an ***instance*** of the type, the latter known as an ***object***. The special value `null` is compatible with all reference types and indicates the absence of an instance.

4.2.1 Class Types

A class type defines a data structure that contains data members (constants and fields), function members (methods, properties, events, indexers, operators, instance constructors, destructors, and static constructors), and nested types. Class types support inheritance, a mechanism whereby derived classes can extend and specialize base classes. Instances of class types are created using *object-creation-expressions* (§7.6.10.1).

Class types are described in §10.

www.it-ebooks.info

Certain predefined class types have special meaning in the C# language, as described in the table below.

Class Type	Description
`System.Object`	The ultimate base class of all other types. (See §4.2.2.)
`System.String`	The string type of the C# language. (See §4.2.3.)
`System.ValueType`	The base class of all value types. (See §4.1.1.)
`System.Enum`	The base class of all enum types. (See §14.)
`System.Array`	The base class of all array types. (See §12.)
`System.Delegate`	The base class of all delegate types. (See §15.)
`System.Exception`	The base class of all exception types. (See §16.)

4.2.2 The object Type

The `object` class type is the ultimate base class of all other types. Every type in C# directly or indirectly derives from the `object` class type.

The keyword `object` is simply an alias for the predefined class `System.Object`.

4.2.3 The dynamic Type

The `dynamic` type, like `object`, can reference any object. When operators are applied to expressions of type `dynamic`, their resolution is deferred until the program is run. Thus, if the operator cannot legally be applied to the referenced object, no error is given during compilation. Instead, an exception will be thrown when resolution of the operator fails at runtime.

The dynamic type is further described in §4.7, and dynamic binding in §7.2.2.

4.2.4 The string Type

The `string` type is a sealed class type that inherits directly from `object`. Instances of the `string` class represent Unicode character strings.

Values of the `string` type can be written as string literals (§2.4.4.5).

The keyword `string` is simply an alias for the predefined class `System.String`.

www.it-ebooks.info

4.2.5 Interface Types

An interface defines a contract. A class or struct that implements an interface must adhere to its contract. An interface may inherit from multiple base interfaces, and a class or struct may implement multiple interfaces.

Interface types are described in §13.

4.2.6 Array Types

An array is a data structure that contains zero or more variables that are accessed through computed indices. The variables contained in an array, also called the elements of the array, are all of the same type, and this type is called the element type of the array.

Array types are described in §12.

4.2.7 Delegate Types

A delegate is a data structure that refers to one or more methods. For instance methods, it also refers to their corresponding object instances.

The closest equivalent of a delegate in C or C++ is a function pointer, but whereas a function pointer can reference only static functions, a delegate can reference both static and instance methods. In the latter case, the delegate stores not only a reference to the method's entry point, but also a reference to the object instance on which to invoke the method.

Delegate types are described in §15.

> **CHRIS SELLS** Although C++ can reference instance member functions via a member function pointer, it's such a difficult thing to get right that the feature might as well be illegal!

4.3 Boxing and Unboxing

The concept of boxing and unboxing is central to C#'s type system. It provides a bridge between *value-type*s and *reference-type*s by permitting any value of a *value-type* to be converted to and from type `object`. Boxing and unboxing enables a unified view of the type system wherein a value of any type can ultimately be treated as an object.

www.it-ebooks.info

JOSEPH ALBAHARI Prior to C# 2.0, boxing and unboxing were the primary means by which you could write a general-purpose collection, such as a list, stack, or queue. Since the introduction of C# 2.0, generics have provided an alternative solution in these cases, which leads to better static type safety and performance. Boxing/unboxing necessarily demands a small performance overhead, because it means copying values, dealing with indirection, and allocating memory on the heap.

JESSE LIBERTY I would go further and say that the introduction of generics has, for all practical purposes, dislodged boxing and unboxing from a central concern to a peripheral one, of interest only when passing value types as `out` or `ref` parameters.

CHRISTIAN NAGEL The normally small performance overhead associated with boxing and unboxing can become huge if you are iterating over large collections. Generic collection classes help with this problem.

4.3.1 Boxing Conversions

A boxing conversion permits a *value-type* to be implicitly converted to a *reference-type*. The following boxing conversions exist:

- From any *value-type* to the type `object`.
- From any *value-type* to the type `System.ValueType`.
- From any *non-nullable-value-type* to any *interface-type* implemented by the *value-type*.
- From any *nullable-type* to any *interface-type* implemented by the underlying type of the *nullable-type*.

VLADIMIR RESHETNIKOV The *nullable-type* does not **implement** the interfaces from its underlying type; it is simply **convertible** to them. This distinction is important in some contexts—for example, in checking generic constraints.

- From any *enum-type* to the type `System.Enum`.
- From any *nullable-type* with an underlying *enum-type* to the type `System.Enum`.

www.it-ebooks.info

BILL WAGNER The choice of the word "conversion" here is illustrative of the behavior seen in these circumstances. You are not reinterpreting the same storage as a different type; you are converting it. That is, you are examining different storage, not looking at the same storage through two different variable types.

Note that an implicit conversion from a type parameter will be executed as a boxing conversion if at runtime it ends up converting from a value type to a reference type (§6.1.10).

Boxing a value of a *non-nullable-value-type* consists of allocating an object instance and copying the *non-nullable-value-type* value into that instance.

Boxing a value of a *nullable-type* produces a null reference if it is the `null` value (`HasValue` is `false`), or the result of unwrapping and boxing the underlying value otherwise.

The actual process of boxing a value of a *non-nullable-value-type* is best explained by imagining the existence of a generic ***boxing class***, which behaves as if it were declared as follows:

```
sealed class Box<T>: System.ValueType
{
    T value;

    public Box(T t) {
        value = t;
    }
}
```

Boxing of a value `v` of type `T` now consists of executing the expression `new Box<T>(v)` and returning the resulting instance as a value of type `object`. Thus the statements

```
int i = 123;
object box = i;
```

conceptually correspond to

```
int i = 123;
object box = new Box<int>(i);
```

A boxing class like `Box<T>` above doesn't actually exist, and the dynamic type of a boxed value isn't actually a class type. Instead, a boxed value of type `T` has the dynamic type `T`, and a dynamic type check using the `is` operator can simply reference type `T`. For example,

```
int i = 123;
object box = i;
if (box is int) {
    Console.Write("Box contains an int");
}
```

will output the string "`Box contains an int`" on the console.

www.it-ebooks.info

A boxing conversion implies *making a copy* of the value being boxed. This is different from a conversion of a *reference-type* to type `object`, in which the value continues to reference the same instance and simply is regarded as the less derived type `object`. For example, given the declaration

```
struct Point
{
    public int x, y;
    public Point(int x, int y) {

        this.x = x;
        this.y = y;
    }
}
```

the following statements

```
Point p = new Point(10, 10);
object box = p;
p.x = 20;
Console.Write(((Point)box).x);
```

will output the value 10 on the console because the implicit boxing operation that occurs in the assignment of `p` to `box` causes the value of `p` to be copied. Had `Point` been declared a `class` instead, the value 20 would be output because `p` and `box` would reference the same instance.

> **ERIC LIPPERT** This possibility is just one reason why it is a good practice to make structs immutable. If the struct cannot mutate, then the fact that boxing makes a copy is irrelevant: Both copies will be identical forever.

4.3.2 Unboxing Conversions

An unboxing conversion permits a *reference-type* to be explicitly converted to a *value-type*. The following unboxing conversions exist:

- From the type `object` to any *value-type*.
- From the type `System.ValueType` to any *value-type*.
- From any *interface-type* to any *non-nullable-value-type* that implements the *interface-type*.
- From any *interface-type* to any *nullable-type* whose underlying type implements the *interface-type*.
- From the type `System.Enum` to any *enum-type*.
- From the type `System.Enum` to any *nullable-type* with an underlying *enum-type*.

www.it-ebooks.info

BILL WAGNER As with boxing, unboxing involves a conversion. If you box a struct and then unbox it, three different storage locations may result. You most certainly do not have three variables examining the same storage.

Note that an explicit conversion to a type parameter will be executed as an unboxing conversion if at runtime it ends up converting from a reference type to a value type (§6.2.6).

An unboxing operation to a *non-nullable-value-type* consists of first checking that the object instance is a boxed value of the given *non-nullable-value-type*, and then copying the value out of the instance.

ERIC LIPPERT Although it is legal to convert an unboxed `int` to an unboxed `double`, it is not legal to convert a boxed `int` to an unboxed `double`—only to an unboxed `int`. This constraint exists because the unboxing instruction would then have to know all the rules for type conversions that are normally done by the compiler. If you need to do these kinds of conversions at runtime, use the `Convert` class instead of an unboxing cast.

Unboxing to a *nullable-type* produces the null value of the *nullable-type* if the source operand is `null`, or the wrapped result of unboxing the object instance to the underlying type of the *nullable-type* otherwise.

Referring to the imaginary boxing class described in the previous section, an unboxing conversion of an object `box` to a *value-type* `T` consists of executing the expression `((Box<T>)box).value`. Thus the statements

```
object box = 123;
int i = (int)box;
```

conceptually correspond to

```
object box = new Box<int>(123);
int i = ((Box<int>)box).value;
```

For an unboxing conversion to a given *non-nullable-value-type* to succeed at runtime, the value of the source operand must be a reference to a boxed value of that *non-nullable-value-type*. If the source operand is `null`, a `System.NullReferenceException` is thrown. If the source operand is a reference to an incompatible object, a `System.InvalidCastException` is thrown.

www.it-ebooks.info

JON SKEET Some unboxing conversions aren't guaranteed to work by the C# specification, yet are legal under the CLI specification. For example, the previously given description precludes unboxing from an enum value to its underlying type, and vice versa:

```
object o = System.DayOfWeek.Sunday;
int i = (int) o;
```

This conversion will succeed in .NET, but would not be guaranteed to succeed on a different C# implementation.

For an unboxing conversion to a given *nullable-type* to succeed at runtime, the value of the source operand must be either `null` or a reference to a boxed value of the underlying *non-nullable-value-type* of the *nullable-type*. If the source operand is a reference to an incompatible object, a `System.InvalidCastException` is thrown.

CHRIS SELLS Boxing and unboxing are designed such that you almost never have to think about them unless you're trying to reduce your memory usage (in which case, profiling is your friend!). However, if you see `out` or `ref` values whose values don't seem to be set properly at the caller's site, suspect boxing.

4.4 Constructed Types

A generic type declaration, by itself, denotes an ***unbound generic type*** that is used as a "blueprint" to form many different types, by way of applying ***type arguments***. The type arguments are written within angle brackets (< and >) immediately following the name of the generic type. A type that includes at least one type argument is called a ***constructed type***. A constructed type can be used in most places in the language in which a type name can appear. An unbound generic type can be used only within a *typeof-expression* (§7.6.11).

Constructed types can also be used in expressions as simple names (§7.6.2) or when accessing a member (§7.6.4).

When a *namespace-or-type-name* is evaluated, only generic types with the correct number of type parameters are considered. Thus it is possible to use the same identifier to identify different types, as long as the types have different numbers of type parameters. This is useful when mixing generic and nongeneric classes in the same program:

www.it-ebooks.info

```
namespace Widgets
{
    class Queue {...}
    class Queue<TElement> {...}
}

namespace MyApplication
{
    using Widgets;

    class X
    {
        Queue q1;               // Nongeneric Widgets.Queue
        Queue<int> q2;          // Generic Widgets.Queue
    }
}
```

A *type-name* might identify a constructed type even though it doesn't specify type parameters directly. This can occur where a type is nested within a generic class declaration, and the instance type of the containing declaration is implicitly used for name lookup (§10.3.8.6):

```
class Outer<T>
{
    public class Inner {...}

    public Inner i;                     // Type of i is Outer<T>.Inner
}
```

In unsafe code, a constructed type cannot be used as an *unmanaged-type* (§18.2).

4.4.1 Type Arguments

Each argument in a type argument list is simply a *type*.

type-argument-list:
    < *type-arguments* >

type-arguments:
    type-argument
    type-arguments , *type-argument*

type-argument:
    type

In unsafe code (§18), a *type-argument* may not be a pointer type. Each type argument must satisfy any constraints on the corresponding type parameter (§10.1.5).

www.it-ebooks.info

4.4.2 Open and Closed Types

All types can be classified as either ***open types*** or ***closed types***. An open type is a type that involves type parameters. More specifically:

- A type parameter defines an open type.
- An array type is an open type if and only if its element type is an open type.
- A constructed type is an open type if and only if one or more of its type arguments is an open type. A constructed nested type is an open type if and only if one or more of its type arguments or the type arguments of its containing type(s) is an open type.

A closed type is a type that is not an open type.

At runtime, all of the code within a generic type declaration is executed in the context of a closed constructed type that was created by applying type arguments to the generic declaration. Each type parameter within the generic type is bound to a particular runtime type. The runtime processing of all statements and expressions always occurs with closed types, and open types occur only during compile-time processing.

Each closed constructed type has its own set of static variables, which are not shared with any other closed constructed types. Since an open type does not exist at runtime, there are no static variables associated with an open type. Two closed constructed types are the same type if they are constructed from the same unbound generic type, and their corresponding type arguments are the same type.

4.4.3 Bound and Unbound Types

The term ***unbound type*** refers to a nongeneric type or an unbound generic type. The term ***bound type*** refers to a nongeneric type or a constructed type.

> **ERIC LIPPERT** Yes, nongeneric types are considered to be *both* bound *and* unbound.

An unbound type refers to the entity declared by a type declaration. An unbound generic type is not itself a type, and it cannot be used as the type of a variable, argument, or return value, or as a base type. The only construct in which an unbound generic type can be referenced is the `typeof` expression (§7.6.11).

4.4.4 Satisfying Constraints

Whenever a constructed type or generic method is referenced, the supplied type arguments are checked against the type parameter constraints declared on the generic type or

www.it-ebooks.info

method (§10.1.5). For each `where` clause, the type argument `A` that corresponds to the named type parameter is checked against each constraint as follows:

- If the constraint is a class type, an interface type, or a type parameter, let `C` represent that constraint with the supplied type arguments substituted for any type parameters that appear in the constraint. To satisfy the constraint, it must be the case that type `A` is convertible to type `C` by one of the following:
  - An identity conversion (§6.1.1).
  - An implicit reference conversion (§6.1.6).
  - A boxing conversion (§6.1.7), provided that type A is a non-nullable value type.
  - An implicit reference, boxing, or type parameter conversion from a type parameter `A` to `C`.
- If the constraint is the reference type constraint (`class`), the type `A` must satisfy one of the following:
  - `A` is an interface type, class type, delegate type, or array type. Both `System.ValueType` and `System.Enum` are reference types that satisfy this constraint.
  - `A` is a type parameter that is known to be a reference type (§10.1.5).
- If the constraint is the value type constraint (`struct`), the type `A` must satisfy one of the following:
  - `A` is a struct type or enum type, but not a nullable type. Both `System.ValueType` and `System.Enum` are reference types that do not satisfy this constraint.
  - `A` is a type parameter having the value type constraint (§10.1.5).
- If the constraint is the constructor constraint `new()`, the type `A` must not be `abstract` and must have a public parameterless constructor. This is satisfied if one of the following is true:
  - `A` is a value type, since all value types have a public default constructor (§4.1.2).
  - `A` is a type parameter having the constructor constraint (§10.1.5).
  - `A` is a type parameter having the value type constraint (§10.1.5).
  - `A` is a class that is not `abstract` and contains an explicitly declared `public` constructor with no parameters.
  - `A` is not `abstract` and has a default constructor (§10.11.4).

A compile-time error occurs if one or more of a type parameter's constraints are not satisfied by the given type arguments.

www.it-ebooks.info

Since type parameters are not inherited, constraints are never inherited either. In the example below, `D` needs to specify the constraint on its type parameter `T` so that `T` satisfies the constraint imposed by the base class `B<T>`. In contrast, class `E` need not specify a constraint, because `List<T>` implements `IEnumerable` for any `T`.

```
class B<T> where T: IEnumerable {...}

class D<T>: B<T> where T: IEnumerable {...}

class E<T>: B<List<T>> {...}
```

4.5 Type Parameters

A type parameter is an identifier designating a value type or reference type that the parameter is bound to at runtime.

type-parameter:
    identifier

Since a type parameter can be instantiated with many different actual type arguments, type parameters have slightly different operations and restrictions than other types:

- A type parameter cannot be used directly to declare a base class (§10.2.4) or interface (§13.1.3).
- The rules for member lookup on type parameters depend on the constraints, if any, applied to the type parameter. They are detailed in §7.4.
- The available conversions for a type parameter depend on the constraints, if any, applied to the type parameter. They are detailed in §6.1.10 and §6.2.6.
- The literal `null` cannot be converted to a type given by a type parameter, except if the type parameter is known to be a reference type (§6.1.10). However, a `default` expression (§7.6.13) can be used instead. In addition, a value with a type given by a type parameter *can* be compared with `null` using `==` and `!=` (§7.10.6) unless the type parameter has the value type constraint.
- A `new` expression (§7.6.10.1) can be used with a type parameter only if the type parameter is constrained by a *constructor-constraint* or the value type constraint (§10.1.5).
- A type parameter cannot be used anywhere within an attribute.
- A type parameter cannot be used in a member access (§7.6.4) or type name (§3.8) to identify a static member or a nested type.
- In unsafe code, a type parameter cannot be used as an *unmanaged-type* (§18.2).

www.it-ebooks.info

As a type, type parameters are purely a compile-time construct. At runtime, each type parameter is bound to a runtime type that was specified by supplying a type argument to the generic type declaration. Thus the type of a variable declared with a type parameter will, at runtime, be a closed constructed type (§4.4.2). The runtime execution of all statements and expressions involving type parameters uses the actual type that was supplied as the type argument for that parameter.

4.6 Expression Tree Types

Expression trees permit anonymous functions to be represented as data structures instead of executable code. Expression trees are values of ***expression tree types*** of the form `System.Linq.Expressions.Expression<D>`, where `D` is any delegate type. For the remainder of this specification, we will refer to these types using the shorthand `Expression<D>`.

If a conversion exists from an anonymous function to a delegate type `D`, a conversion also exists to the expression tree type `Expression<D>`. Whereas the conversion of an anonymous function to a delegate type generates a delegate that references executable code for the anonymous function, conversion to an expression tree type creates an expression tree representation of the anonymous function.

Expression trees are efficient in-memory data representations of anonymous functions and make the structure of the anonymous function transparent and explicit.

Just like a delegate type `D`, `Expression<D>` is said to have parameter and return types, which are the same as those of `D`.

The following example represents an anonymous function both as executable code and as an expression tree. Because a conversion exists to `Func<int,int>`, a conversion also exists to `Expression<Func<int,int>>`:

```
Func<int,int> del = x => x + 1;                 // Code
Expression<Func<int,int>> exp = x => x + 1;     // Data
```

Following these assignments, the delegate `del` references a method that returns `x + 1`, and the expression tree `exp` references a data structure that describes the expression `x => x + 1`.

The exact definition of the generic type `Expression<D>` as well as the precise rules for constructing an expression tree when an anonymous function is converted to an expression tree type are implementation defined.

www.it-ebooks.info

Two things are important to make explicit:

- Not all anonymous functions can be represented as expression trees. For instance, anonymous functions with statement bodies and anonymous functions containing assignment expressions cannot be represented. In these cases, a conversion still exists, but will fail at compile time.
- `Expression<D>` offers an instance method `Compile` that produces a delegate of type `D`:

```
Func<int,int> del2 = exp.Compile();
```

Invoking this delegate causes the code represented by the expression tree to be executed. Thus, given the definitions above, `del` and `del2` are equivalent, and the following two statements will have the same effect:

```
int i1 = del(1);
int i2 = del2(1);
```

After executing this code, `i1` and `i2` will both have the value `2`.

4.7 The dynamic Type

The type `dynamic` has special meaning in C#. Its purpose is to allow dynamic binding, which is described in detail in §7.2.2.

The `dynamic` type is considered identical to the `object` type except in the following respects:

- Operations on expressions of type `dynamic` can be dynamically bound (§7.2.2).
- Type inference (§7.5.2) will prefer `dynamic` over `object` if both are candidates.

Because of this equivalence, the following statements hold:

- There is an implicit identity conversion between `object` and `dynamic`, and between constructed types that are the same when replacing `dynamic` with `object`.
- Implicit and explicit conversions to and from `object` also apply to and from `dynamic`.
- Method signatures that are the same when replacing `dynamic` with `object` are considered the same signature.

The type `dynamic` is indistinguishable from `object` at runtime.

An expression of the type `dynamic` is referred to as a ***dynamic expression***.

www.it-ebooks.info

ERIC LIPPERT The type `dynamic` is a bizarre type, but it is important to note that, from the compiler's perspective, it *is* a type. Unlike with `var`, you can use it in most of the situations that call for a type: return types, parameter types, type arguments, and so on.

PETER SESTOFT Actually, `var` is a reserved word, not a compile-time type, whereas `dynamic` is a compile-time type. The `var` keyword tells the compiler, "Please infer the compile-time type of this variable from its initializer expression." The `dynamic` type essentially tells the compiler, "Do not worry about compile-time type checking of expressions in which this variable appears; the runtime system will do the right thing based on the runtime type of the *value* of the variable (or throw an exception, where the compiler would have reported a type error)." The type `dynamic` cannot be used as receiver (`this` type) of an extension method, as base type of a class, or as type bound for a generic type parameter, but otherwise it can be used pretty much like any other type.

MAREK SAFAR Method signatures are considered to be same when using the `dynamic` and `object` types. This allows use of a nice trick: The interface method declared using type `object` can be directly implemented using a method with type `dynamic`.

CHRIS SELLS I begin to wonder about any language where the following string of characters is both valid and meaningful:

```
class Foo {
  public static dynamic DoFoo() {...}
}
```

Of course, this means that the `DoFoo` method is a type method (as opposed to an instance method) and that the type of the return value is unknown until runtime, but it's hard not to read `DoFoo` as both `static` and `dynamic` at the same time and worry about an occurrence of a singularity.

www.it-ebooks.info

This page intentionally left blank

www.it-ebooks.info

5. Variables

Variables represent storage locations. Every variable has a type that determines what values can be stored in the variable. C# is a type-safe language, and the C# compiler guarantees that values stored in variables are always of the appropriate type. The value of a variable can be changed through assignment or through use of the ++ and -- operators.

A variable must be ***definitely assigned*** (§5.3) before its value can be obtained.

As described in the following sections, variables are either ***initially assigned*** or ***initially unassigned***. An initially assigned variable has a well-defined initial value and is always considered definitely assigned. An initially unassigned variable has no initial value. For an initially unassigned variable to be considered definitely assigned at a certain location, an assignment to the variable must occur in every possible execution path leading to that location.

5.1 Variable Categories

C# defines seven categories of variables: static variables, instance variables, array elements, value parameters, reference parameters, output parameters, and local variables. The sections that follow describe each of these categories.

In the example

```
class A
{
    public static int x;
    int y;

    void F(int[] v, int a, ref int b, out int c)
    {
        int i = 1;
        c = a + b++;
    }
}
```

`x` is a static variable, `y` is an instance variable, `v[0]` is an array element, `a` is a value parameter, `b` is a reference parameter, `c` is an output parameter, and `i` is a local variable.

www.it-ebooks.info

JESSE LIBERTY It is inevitable—but unfortunate—that x is used as the name of the first variable described in all our books. Variable names should be self-revealing and, except perhaps when used as counters in for loops, should never be given single-letter names. I'd much prefer to see this example written (to stretch the point) as follows:

```
class ASimpleExampleClass
{
    public static int staticMember;
    int memberVariable;
    void ExampleFunction(
        int[] arrayOfIntsParam,
        int simpleParam,
        ref int refParam,
        out int OutParam)
    {
        int tempVariable = 1;
        outParam = simpleParam + refParam ++;
    }
}
```

While the naming scheme may seem cumbersome in this simple example, there is no ambiguity about what is happening or why—and the explanation that follows is nearly superfluous.

5.1.1 Static Variables

A field declared with the `static` modifier is called a ***static variable***. A static variable comes into existence before execution of the static constructor (§10.12) for its containing type, and ceases to exist when the associated application domain ceases to exist.

The initial value of a static variable is the default value (§5.2) of the variable's type.

For purposes of definite assignment checking, a static variable is considered initially assigned.

5.1.2 Instance Variables

A field declared without the `static` modifier is called an ***instance variable***.

5.1.2.1 *Instance Variables in Classes*

An instance variable of a class comes into existence when a new instance of that class is created, and ceases to exist when there are no references to that instance and the instance's destructor (if any) has executed.

The initial value of an instance variable of a class is the default value (§5.2) of the variable's type.

www.it-ebooks.info

For the purpose of definite assignment checking, an instance variable of a class is considered initially assigned.

5.1.2.2 *Instance Variables in Structs*

An instance variable of a struct has exactly the same lifetime as the struct variable to which it belongs. In other words, when a variable of a struct type comes into existence or ceases to exist, so, too, do the instance variables of the struct.

The initial assignment state of an instance variable of a struct is the same as that of the containing struct variable. In other words, when a struct variable is considered initially assigned, so, too, are its instance variables. When a struct variable is considered initially unassigned, its instance variables are likewise unassigned.

> **BILL WAGNER** An instance variable in a struct of reference type may not be eligible for garbage collection when the struct containing it ceases to exist. If the object is reachable in another path, the object is still alive even though the instance variable of the struct is not. This is also true for array elements.

5.1.3 Array Elements

The elements of an array come into existence when an array instance is created, and cease to exist when there are no references to that array instance.

The initial value of each of the elements of an array is the default value (§5.2) of the type of the array elements.

For the purpose of definite assignment checking, an array element is considered initially assigned.

5.1.4 Value Parameters

A parameter declared without a `ref` or `out` modifier is a ***value parameter***.

A value parameter comes into existence upon invocation of the function member (method, instance constructor, accessor, or operator) or anonymous function to which the parameter belongs, and is initialized with the value of the argument given in the invocation. A value parameter normally ceases to exist upon return of the function member or anonymous function. However, if the value parameter is captured by an anonymous function (§7.15), its lifetime extends at least until the delegate or expression tree created from that anonymous function is eligible for garbage collection.

For the purpose of definite assignment checking, a value parameter is considered initially assigned.

www.it-ebooks.info

5.1.5 Reference Parameters

A parameter declared with a `ref` modifier is a ***reference parameter***.

A reference parameter does not create a new storage location. Instead, a reference parameter represents the same storage location as the variable given as the argument in the function member or anonymous function invocation. Thus the value of a reference parameter is always the same as the underlying variable.

The following definite assignment rules apply to reference parameters. Note the different rules for output parameters described in §5.1.6.

- A variable must be definitely assigned (§5.3) before it can be passed as a reference parameter in a function member or delegate invocation.
- Within a function member or anonymous function, a reference parameter is considered initially assigned.

Within an instance method or instance accessor of a struct type, the `this` keyword behaves exactly as a reference parameter of the struct type (§7.6.7).

ERIC LIPPERT Less formally, the difference between a reference and an output parameter is that a reference parameter represents an "input and output" parameter: It must be initialized when the method starts, and the method may optionally change the contents. An output parameter, by comparison, is used for output; the method must fill in the contents and may not peek at the contents until after it has done so.

JON SKEET Even an output parameter doesn't have to be used *solely* for output. In particular, after a method has assigned a value to an output parameter, it can then read from it. Because the parameter will share a storage location with another variable, however, there's no guarantee that the value won't change again after it's been assigned in the method. This approach prevents output parameters being used covariantly.

ERIC LIPPERT Some other programming languages support another kind of parameter passing: an "input only" reference. That is, the reference to the variable is passed to the method but, unlike a C# reference parameter, the method may not write to it—only read from it. This approach is useful for efficiently passing large value types around; passing a reference to a variable may lead to better performance than passing a copy of the value if the value is large. C# does not support this kind of reference passing.

www.it-ebooks.info

5.1.6 Output Parameters

A parameter declared with an `out` modifier is an ***output parameter***.

An output parameter does not create a new storage location. Instead, an output parameter represents the same storage location as the variable given as the argument in the function member or delegate invocation. Thus the value of an output parameter is always the same as the underlying variable.

The following definite assignment rules apply to output parameters. Note the different rules for reference parameters described in §5.1.5.

- A variable need not be definitely assigned before it can be passed as an output parameter in a function member or delegate invocation.
- Following the normal completion of a function member or delegate invocation, each variable that was passed as an output parameter is considered assigned in that execution path.
- Within a function member or anonymous function, an output parameter is considered initially unassigned.
- Every output parameter of a function member or anonymous function must be definitely assigned (§5.3) before the function member or anonymous function returns normally.

Within an instance constructor of a struct type, the `this` keyword behaves exactly as an output parameter of the struct type (§7.6.7).

CHRISTIAN NAGEL Instead of using output parameters to return multiple values from a function, you might use the tuple type. This type is new with .NET 4.

5.1.7 Local Variables

A ***local variable*** is declared either by a *local-variable-declaration*, which may occur in a *block*, a *for-statement*, a *switch-statement*, or a *using-statement*; by a *foreach-statement*; or by a *specific-catch-clause* for a *try-statement*.

The lifetime of a local variable is the portion of program execution during which storage is guaranteed to be reserved for it. This lifetime extends at least from entry into the *block*, *for-statement*, *switch-statement*, *using-statement*, *foreach-statement*, or *specific-catch-clause* with which it is associated, until execution of that *block*, *for-statement*, *switch-statement*, *using-statement*, *foreach-statement*, or *specific-catch-clause* ends in any way. (Entering an enclosed *block* or calling a method suspends, but does not end, execution of the current *block*, *for-statement*, *switch-statement*, *using-statement*, *foreach-statement*, or *specific-catch-clause*.) If the

www.it-ebooks.info

local variable is captured by an anonymous function (§7.15.5.1), its lifetime extends at least until the delegate or expression tree created from the anonymous function, along with any other objects that come to reference the captured variable, are eligible for garbage collection.

BILL WAGNER This last sentence has important implications for the possible cost of local variables captured in anonymous functions. The lifetimes of those variables may be much longer, which means any other objects that are referenced by those local variables will also live much longer.

If the parent *block*, *for-statement*, *switch-statement*, *using-statement*, *foreach-statement*, or *specific-catch-clause* is entered recursively, a new instance of the local variable is created each time, and its *local-variable-initializer*, if any, is evaluated each time.

A local variable introduced by a *local-variable-declaration* is not automatically initialized and thus has no default value. For the purpose of definite assignment checking, a local variable introduced by a *local-variable-declaration* is considered initially unassigned. A *local-variable-declaration* may include a *local-variable-initializer*, in which case the variable is considered definitely assigned only after the initializing expression (§5.3.3.4).

ERIC LIPPERT Requiring local variables to be definitely assigned rather than automatically assigning them to their default values might seem like it confers a performance benefit; after all, the compiler need not generate code that redundantly assigns a default value to the location. In reality, this is not the motivation for the feature. In practice, the CLR does initialize local variables to their default values, which typically takes place very rapidly. The motivating factor for definite assignment checks is that it prevents a common cause of bugs. C# does not guess that you meant for the local variable to be initialized and hide your bug; it requires that the local variable be explicitly initialized before you use it.

Within the scope of a local variable introduced by a *local-variable-declaration*, it is a compile-time error to refer to that local variable in a textual position that precedes its *local-variable-declarator*. If the local variable declaration is implicit (§8.5.1), it is also an error to refer to the variable within its *local-variable-declarator*.

A local variable introduced by a *foreach-statement* or a *specific-catch-clause* is considered definitely assigned in its entire scope.

The actual lifetime of a local variable is implementation-dependent. For example, a compiler might statically determine that a local variable in a block is used for only a small

www.it-ebooks.info

portion of that block. Using this analysis, the compiler could generate code that results in the variable's storage having a shorter lifetime than its containing block.

The storage referred to by a local reference variable is reclaimed independently of the lifetime of that local reference variable (§3.9).

CHRIS SELLS C required variables to be declared at the top of a scope:

```
void F() {
    int x = ...;
    ...
    Foo(x); // What was x again?
    ...
}
```

In C#, it's generally considered bad practice to declare a variable far from where it's used. Instead, the following form is preferred:

```
void F() {
    ...
    int x = ...;
    Foo(x); // Oh, right, I see x...
    ...
}
```

Taking this principle, which is known as "locality of reference," into account makes your programs more readable and more maintainable.

5.2 Default Values

The following categories of variables are automatically initialized to their default values:

- Static variables.
- Instance variables of class instances.
- Array elements.

The default value of a variable depends on the type of the variable and is determined as follows:

- For a variable of a *value-type*, the default value is the same as the value computed by the *value-type*'s default constructor (§4.1.2).
- For a variable of a *reference-type*, the default value is `null`.

www.it-ebooks.info

Initialization to default values is typically done by having the memory manager or garbage collector initialize memory to all-bits-zero before it is allocated for use. For this reason, it is convenient to use all-bits-zero to represent the null reference.

5.3 Definite Assignment

At a given location in the executable code of a function member, a variable is said to be ***definitely assigned*** if the compiler can prove, by a particular static flow analysis (§5.3.3), that the variable has been automatically initialized or has been the target of at least one assignment. Informally stated, the rules of definite assignment are:

- An initially assigned variable (§5.3.1) is always considered definitely assigned.
- An initially unassigned variable (§5.3.2) is considered definitely assigned at a given location if all possible execution paths leading to that location contain at least one of the following:
  - A simple assignment (§7.17.1) in which the variable is the left operand.
  - An invocation expression (§7.6.5) or object creation expression (§7.6.10.1) that passes the variable as an output parameter.
  - For a local variable, a local variable declaration (§8.5.1) that includes a variable initializer.

The formal specification underlying the above informal rules is described in §5.3.1, §5.3.2, and §5.3.3.

The definite assignment states of instance variables of a *struct-type* variable are tracked individually as well as collectively. In addition to the rules above, the following rules apply to *struct-type* variables and their instance variables:

- An instance variable is considered definitely assigned if its containing *struct-type* variable is considered definitely assigned.
- A *struct-type* variable is considered definitely assigned if each of its instance variables is considered definitely assigned.

Definite assignment is a requirement in the following contexts:

- A variable must be definitely assigned at each location where its value is obtained. This ensures that undefined values never occur. The occurrence of a variable in an expression is considered to obtain the value of the variable, except when

www.it-ebooks.info

  - The variable is the left operand of a simple assignment,
  - The variable is passed as an output parameter, or
  - The variable is a *struct-type* variable and occurs as the left operand of a member access.
- A variable must be definitely assigned at each location where it is passed as a reference parameter. This ensures that the function member being invoked can consider the reference parameter initially assigned.
- All output parameters of a function member must be definitely assigned at each location where the function member returns (through a `return` statement or through execution reaching the end of the function member body). This ensures that function members do not return undefined values in output parameters, thus enabling the compiler to consider a function member invocation that takes a variable as an output parameter equivalent to an assignment to the variable.
- The `this` variable of a *struct-type* instance constructor must be definitely assigned at each location where that instance constructor returns.

5.3.1 Initially Assigned Variables

The following categories of variables are classified as initially assigned:

- Static variables.
- Instance variables of class instances.
- Instance variables of initially assigned struct variables.
- Array elements.
- Value parameters.
- Reference parameters.
- Variables declared in a `catch` clause or a `foreach` statement.

5.3.2 Initially Unassigned Variables

The following categories of variables are classified as initially unassigned:

- Instance variables of initially unassigned struct variables.
- Output parameters, including the `this` variable of struct instance constructors.
- Local variables, except those declared in a `catch` clause or a `foreach` statement.

www.it-ebooks.info

5.3.3 Precise Rules for Determining Definite Assignment

To determine that each used variable is definitely assigned, the compiler must use a process that is equivalent to the one described in this section.

The compiler processes the body of each function member that has one or more initially unassigned variables. For each initially unassigned variable v, the compiler determines a ***definite assignment state*** for v at each of the following points in the function member:

- At the beginning of each statement.
- At the end point (§8.1) of each statement.
- On each arc that transfers control to another statement or to the end point of a statement.
- At the beginning of each expression.
- At the end of each expression.

The definite assignment state of v can be either:

- Definitely assigned. This indicates that on all possible control flows to this point, v has been assigned a value.
- Not definitely assigned. For the state of a variable at the end of an expression of type `bool`, the state of a variable that isn't definitely assigned may (but doesn't necessarily) fall into one of the following substates:
  - Definitely assigned after true expression. This state indicates that v is definitely assigned if the boolean expression evaluated as true, but is not necessarily assigned if the boolean expression evaluated as false.
  - Definitely assigned after false expression. This state indicates that v is definitely assigned if the boolean expression evaluated as false, but is not necessarily assigned if the boolean expression evaluated as true.

The following rules govern how the state of a variable v is determined at each location.

5.3.3.1 *General Rules for Statements*

- v is not definitely assigned at the beginning of a function member body.
- v is definitely assigned at the beginning of any unreachable statement.

www.it-ebooks.info

ERIC LIPPERT This rule may strike you as unusual. Why should every variable be considered to be definitely assigned within an unreachable statement? Some reasons are discussed here.

If the statement is unreachable, then it will not execute. If it will not execute, then it will not read from an unassigned variable. Therefore, it is not a problem; the compiler is looking for potential problems, and this is not one.

Also, unreachable statements are almost certainly mistakes. Once the compiler has identified one mistake, odds are good that a whole host of related mistakes are clustered around it. Rather than reporting a huge number of related problems, it is often a better idea to report just one problem and allow the user to fix it, rather than trying to report every possible specification violation.

- The definite assignment state of *v* at the beginning of any other statement is determined by checking the definite assignment state of *v* on all control flow transfers that target the beginning of that statement. If (and only if) *v* is definitely assigned on all such control flow transfers, then *v* is definitely assigned at the beginning of the statement. The set of possible control flow transfers is determined in the same way as for checking statement reachability (§8.1).
- The definite assignment state of *v* at the end point of a block, `checked`, `unchecked`, `if`, `while`, `do`, `for`, `foreach`, `lock`, `using`, or `switch` statement is determined by checking the definite assignment state of *v* on all control flow transfers that target the end point of that statement. If *v* is definitely assigned on all such control flow transfers, then *v* is definitely assigned at the end point of the statement. Otherwise, *v* is not definitely assigned at the end point of the statement. The set of possible control flow transfers is determined in the same way as for checking statement reachability (§8.1).

5.3.3.2 *Block Statements, checked Statements, and unchecked Statements*

The definite assignment state of *v* on the control transfer to the first statement of the statement list in the block (or to the end point of the block, if the statement list is empty) is the same as the definite assignment statement of *v* before the block, `checked`, or `unchecked` statement.

5.3.3.3 *Expression Statements*

For an expression statement *stmt* that consists of the expression *expr*:

- *v* has the same definite assignment state at the beginning of *expr* as at the beginning of *stmt*.
- If *v* is definitely assigned at the end of *expr*, it is definitely assigned at the end point of *stmt*; otherwise, it is not definitely assigned at the end point of *stmt*.

www.it-ebooks.info

5.3.3.4 *Declaration Statements*

- If *stmt* is a declaration statement without initializers, then *v* has the same definite assignment state at the end point of *stmt* as at the beginning of *stmt*.
- If *stmt* is a declaration statement with initializers, then the definite assignment state for *v* is determined as if *stmt* were a statement list, with one assignment statement for each declaration with an initializer (in the order of declaration).

5.3.3.5 *if Statements*

For an `if` statement *stmt* of the form:

`if (` *expr* `)` *then-stmt* `else` *else-stmt*

- *v* has the same definite assignment state at the beginning of *expr* as at the beginning of *stmt*.
- If *v* is definitely assigned at the end of *expr*, then it is definitely assigned on the control flow transfer to *then-stmt* and to either *else-stmt* or to the end point of *stmt* if there is no *else* clause.
- If *v* has the state "definitely assigned after true expression" at the end of *expr*, then it is definitely assigned on the control flow transfer to *then-stmt*, and not definitely assigned on the control flow transfer to either *else-stmt* or to the end point of *stmt* if there is no *else* clause.
- If *v* has the state "definitely assigned after false expression" at the end of *expr*, then it is definitely assigned on the control flow transfer to *else-stmt*, and not definitely assigned on the control flow transfer to *then-stmt*. It is definitely assigned at the end point of *stmt* if and only if it is definitely assigned at the end point of *then-stmt*.
- Otherwise, *v* is considered not definitely assigned on the control flow transfer to either *then-stmt* or *else-stmt*, or to the end point of *stmt* if there is no *else* clause.

5.3.3.6 *switch Statements*

In a `switch` statement *stmt* with a controlling expression *expr*:

- The definite assignment state of *v* at the beginning of *expr* is the same as the state of *v* at the beginning of *stmt*.
- The definite assignment state of *v* on the control flow transfer to a reachable `switch` block statement list is the same as the definite assignment state of *v* at the end of *expr*.

www.it-ebooks.info

ERIC LIPPERT Unfortunately, in the few cases where it is feasible to switch exhaustively on the entire range of a controlling expression without a default `switch` label (that is, `bool`, the byte types, and their corresponding nullable types), definite assignment checking does not take into consideration the possibility that a variable may be definitely assigned at the end of the `switch` if it is definitely assigned at the end of every `switch` section. For example:

```
int x;
bool b = B();
switch(b) {
case true : x = 1; break;
case false: x = 2; break;
}
Console.WriteLine(x); // Error: x is not definitely assigned
```

In those rare cases, you can always put an unnecessary default `switch` label on one of the `switch` sections to make the definite assignment checker happy.

5.3.3.7 *while Statements*

For a `while` statement *stmt* of the form:

```
while ( expr ) while-body
```

- *v* has the same definite assignment state at the beginning of *expr* as at the beginning of *stmt*.
- If *v* is definitely assigned at the end of *expr*, then it is definitely assigned on the control flow transfer to *while-body* and to the end point of *stmt*.
- If *v* has the state "definitely assigned after true expression" at the end of *expr*, then it is definitely assigned on the control flow transfer to *while-body*, but not definitely assigned at the end point of *stmt*.
- If *v* has the state "definitely assigned after false expression" at the end of *expr*, then it is definitely assigned on the control flow transfer to the end point of *stmt*, but not definitely assigned on the control flow transfer to *while-body*.

5.3.3.8 *do Statements*

For a `do` statement *stmt* of the form:

```
do do-body while ( expr ) ;
```

- *v* has the same definite assignment state on the control flow transfer from the beginning of *stmt* to *do-body* as at the beginning of *stmt*.

www.it-ebooks.info

- *v* has the same definite assignment state at the beginning of *expr* as at the end point of *do-body*.
- If *v* is definitely assigned at the end of *expr*, then it is definitely assigned on the control flow transfer to the end point of *stmt*.
- If *v* has the state "definitely assigned after false expression" at the end of *expr*, then it is definitely assigned on the control flow transfer to the end point of *stmt*.

5.3.3.9 *for Statements*

Definite assignment checking for a `for` statement of the form:

```
for ( for-initializer ; for-condition ; for-iterator ) embedded-statement
```

is done as if the statement were written:

```
{
    for-initializer ;
    while ( for-condition ) {
        embedded-statement ;
        for-iterator ;
    }
}
```

If the *for-condition* is omitted from the `for` statement, then evaluation of definite assignment proceeds as if *for-condition* were replaced with `true` in the above expansion.

5.3.3.10 *break, continue, and goto Statements*

The definite assignment state of *v* on the control flow transfer caused by a `break`, `continue`, or `goto` statement is the same as the definite assignment state of *v* at the beginning of the statement.

5.3.3.11 *throw Statements*

For a statement *stmt* of the form:

```
throw expr ;
```

the definite assignment state of *v* at the beginning of *expr* is the same as the definite assignment state of *v* at the beginning of *stmt*.

5.3.3.12 *return Statements*

For a statement *stmt* of the form:

```
return expr ;
```


www.it-ebooks.info

- The definite assignment state of *v* at the beginning of *expr* is the same as the definite assignment state of *v* at the beginning of *stmt*.
- If *v* is an output parameter, then it must be definitely assigned either
  - After *expr* or
  - At the end of the `finally` block of a `try-finally` or `try-catch-finally` that encloses the `return` statement.

For a statement *stmt* of the form:

```
return ;
```

- If *v* is an output parameter, then it must be definitely assigned either
  - Before *stmt* or
  - At the end of the `finally` block of a `try-finally` or `try-catch-finally` that encloses the `return` statement.

5.3.3.13 *try-catch Statements*

> **BILL WAGNER** Starting here, you see how any of the nonstructured statements (e.g., `throw`, `catch`, `goto`, `finally`) can complicate both the compiler's analysis and your own understanding. Be careful how you use these statements in your regular logic. The `try/finally` statement is a simplified special case, but the general cases can greatly decrease program readability.

For a statement *stmt* of the form:

```
try try-block
catch(...) catch-block-1
...
catch(...) catch-block-n
```

- The definite assignment state of *v* at the beginning of *try-block* is the same as the definite assignment state of *v* at the beginning of *stmt*.
- The definite assignment state of *v* at the beginning of *catch-block-i* (for any *i*) is the same as the definite assignment state of *v* at the beginning of *stmt*.
- The definite assignment state of *v* at the end point of *stmt* is definitely assigned if (and only if) *v* is definitely assigned at the end point of *try-block* and every *catch-block-i* (for every *i* from 1 to *n*).

www.it-ebooks.info

5.3.3.14 *try-finally Statements*

For a `try` statement *stmt* of the form:

```
try try-block finally finally-block
```

- The definite assignment state of *v* at the beginning of *try-block* is the same as the definite assignment state of *v* at the beginning of *stmt*.
- The definite assignment state of *v* at the beginning of *finally-block* is the same as the definite assignment state of *v* at the beginning of *stmt*.
- The definite assignment state of *v* at the end point of *stmt* is definitely assigned if (and only if) at least one of the following is true:
  - *v* is definitely assigned at the end point of *try-block.*
  - *v* is definitely assigned at the end point of *finally-block.*

If a control flow transfer (for example, a `goto` statement) is made that begins within *try-block* and ends outside of *try-block*, then *v* is also considered definitely assigned on that control flow transfer if *v* is definitely assigned at the end point of *finally-block*. (This is not an "only if": If *v* is definitely assigned for another reason on this control flow transfer, then it is still considered definitely assigned.)

5.3.3.15 *try-catch-finally Statements*

Definite assignment analysis for a `try-catch-finally` statement of the form:

```
try try-block
catch(...) catch-block-1
...
catch(...) catch-block-n
finally finally-block
```

is done as if the statement were a `try-finally` statement enclosing a `try-catch` statement:

```
try {
    try try-block
    catch(...) catch-block-1
    ...
    catch(...) catch-block-n
}
finally finally-block
```

The following example demonstrates how the different blocks of a `try` statement (§8.10) affect definite assignment.

www.it-ebooks.info

```
class A
{
    static void F()
    {
        int i, j;
        try
        {
            goto LABEL;
            // Neither i nor j definitely assigned
            i = 1;
            // i definitely assigned
        }

        catch
        {
            // Neither i nor j definitely assigned
            i = 3;
            // i definitely assigned
        }

        finally
        {
            // Neither i nor j definitely assigned
            j = 5;
            // j definitely assigned
        }
    // i and j definitely assigned
    LABEL: ;
        // j definitely assigned

    }
}
```

5.3.3.16 *foreach Statements*

For a `foreach` statement *stmt* of the form:

`foreach` `(` *type identifier* `in` *expr* `)` *embedded-statement*

- The definite assignment state of *v* at the beginning of *expr* is the same as the state of *v* at the beginning of *stmt*.
- The definite assignment state of *v* on the control flow transfer to *embedded-statement* or to the end point of *stmt* is the same as the state of *v* at the end of *expr*.

5.3.3.17 *using Statements*

For a `using` statement *stmt* of the form:

`using` `(` *resource-acquisition* `)` *embedded-statement*

- The definite assignment state of *v* at the beginning of *resource-acquisition* is the same as the state of *v* at the beginning of *stmt*.

www.it-ebooks.info

- The definite assignment state of *v* on the control flow transfer to *embedded-statement* is the same as the state of *v* at the end of *resource-acquisition*.

5.3.3.18 *lock Statements*

For a `lock` statement *stmt* of the form:

```
lock ( expr ) embedded-statement
```

- The definite assignment state of *v* at the beginning of *expr* is the same as the state of *v* at the beginning of *stmt*.
- The definite assignment state of *v* on the control flow transfer to *embedded-statement* is the same as the state of *v* at the end of *expr*.

5.3.3.19 *yield Statements*

For a `yield return` statement *stmt* of the form:

```
yield return expr ;
```

- The definite assignment state of *v* at the beginning of *expr* is the same as the state of *v* at the beginning of *stmt*.
- The definite assignment state of *v* at the end of *stmt* is the same as the state of *v* at the end of *expr*.

A `yield break` statement has no effect on the definite assignment state.

5.3.3.20 *General Rules for Simple Expressions*

The following rule applies to these kinds of expressions: literals (§7.6.1), simple names (§7.6.2), member access expressions (§7.6.4), non-indexed base access expressions (§7.6.8), `typeof` expressions (§7.6.11), and default value expressions (§7.6.13).

- The definite assignment state of *v* at the end of such an expression is the same as the definite assignment state of *v* at the beginning of the expression.

5.3.3.21 *General Rules for Expressions with Embedded Expressions*

The following rules apply to these kinds of expressions: parenthesized expressions (§7.6.3); element access expressions (§7.6.6); base access expressions with indexing (§7.6.8); increment and decrement expressions (§7.6.9, §7.7.5); cast expressions (§7.7.6); unary `+`, `-`, `~`, `*` expressions; binary `+`, `-`, `*`, `/`, `%`, `<<`, `>>`, `<`, `<=`, `>`, `>=`, `==`, `!=`, `is`, `as`, `&`, `|`, `^` expressions (§7.8, §7.9, §7.10, §7.11); compound assignment expressions (§7.17.2); `checked` and `unchecked` expressions (§7.6.12); plus array and delegate creation expressions (§7.6.10).

www.it-ebooks.info

Each of these expressions has one or more subexpressions that are unconditionally evaluated in a fixed order. For example, the binary `%` operator evaluates the left-hand side of the operator, then the right-hand side. An indexing operation evaluates the indexed expression, and then evaluates each of the index expressions, in order from left to right. For an expression *expr* that has subexpressions $expr_1$, $expr_2$, ..., $expr_n$, evaluated in that order:

- The definite assignment state of *v* at the beginning of $expr_1$ is the same as the definite assignment state at the beginning of *expr*.
- The definite assignment state of *v* at the beginning of $expr_i$ (*i* greater than 1) is the same as the definite assignment state at the end of $expr_{i-1}$.
- The definite assignment state of *v* at the end of *expr* is the same as the definite assignment state at the end of $expr_n$.

5.3.3.22 *Invocation Expressions and Object Creation Expressions*

For an invocation expression *expr* of the form:

primary-expression `(` arg_1 `,` arg_2 `, ... ,` arg_n `)`

or an object creation expression of the form:

`new` *type* `(` arg_1 `,` arg_2 `, ... ,` arg_n `)`

- For an invocation expression, the definite assignment state of *v* before *primary-expression* is the same as the state of *v* before *expr*.
- For an invocation expression, the definite assignment state of *v* before arg_1 is the same as the state of *v* after *primary-expression*.
- For an object creation expression, the definite assignment state of *v* before arg_1 is the same as the state of *v* before *expr*.
- For each argument arg_i, the definite assignment state of *v* after arg_i is determined by the normal expression rules, ignoring any `ref` or `out` modifiers.
- For each argument arg_i for any *i* greater than 1, the definite assignment state of *v* before arg_i is the same as the state of *v* after arg_{i-1}.
- If the variable *v* is passed as an `out` argument (i.e., an argument of the form "`out` *v*") in any of the arguments, then the state of *v* after *expr* is definitely assigned. Otherwise, the state of *v* after *expr* is the same as the state of *v* after arg_n.
- For array initializers (§7.6.10.4), object initializers (§7.6.10.2), collection initializers (§7.6.10.3), and anonymous object initializers (§7.6.10.6), the definite assignment state is determined by the expansion that these constructs are defined in terms of.

www.it-ebooks.info

5.3.3.23 *Simple Assignment Expressions*

For an expression *expr* of the form *w* = *expr-rhs*:

- The definite assignment state of *v* before *expr-rhs* is the same as the definite assignment state of *v* before *expr*.
- If *w* is the same variable as *v*, then the definite assignment state of *v* after *expr* is definitely assigned. Otherwise, the definite assignment state of *v* after *expr* is the same as the definite assignment state of *v* after *expr-rhs*.

5.3.3.24 *&& Expressions*

For an expression *expr* of the form *expr-first* && *expr-second*:

- The definite assignment state of *v* before *expr-first* is the same as the definite assignment state of *v* before *expr*.
- The definite assignment state of *v* before *expr-second* is definitely assigned if the state of *v* after *expr-first* is either definitely assigned or "definitely assigned after true expression." Otherwise, it is not definitely assigned.
- The definite assignment state of *v* after *expr* is determined by:
  - If the state of *v* after *expr-first* is definitely assigned, then the state of *v* after *expr* is definitely assigned.
  - Otherwise, if the state of *v* after *expr-second* is definitely assigned, and the state of *v* after *expr-first* is "definitely assigned after false expression," then the state of *v* after *expr* is definitely assigned.
  - Otherwise, if the state of *v* after *expr-second* is definitely assigned or "definitely assigned after true expression," then the state of *v* after *expr* is "definitely assigned after true expression."
  - Otherwise, if the state of *v* after *expr-first* is "definitely assigned after false expression," and the state of *v* after *expr-second* is "definitely assigned after false expression," then the state of *v* after *expr* is "definitely assigned after false expression."
  - Otherwise, the state of *v* after *expr* is not definitely assigned.

In the example

```
class A
{
    static void F(int x, int y)
    {
        int i;
        if (x >= 0 && (i = y) >= 0)
```


www.it-ebooks.info

```
        {
            // i definitely assigned
        }
        else
        {
            // i not definitely assigned
        }
        // i not definitely assigned
    }
}
```

the variable `i` is considered definitely assigned in one of the embedded statements of an `if` statement but not in the other. In the `if` statement in method `F`, the variable `i` is definitely assigned in the first embedded statement because execution of the expression `(i = y)` always precedes execution of this embedded statement. In contrast, the variable `i` is not definitely assigned in the second embedded statement, because `x >= 0` might have tested false, resulting in the variable `i` being unassigned.

5.3.3.25 *|| Expressions*

For an expression *expr* of the form *expr-first* `||` *expr-second*:

- The definite assignment state of *v* before *expr-first* is the same as the definite assignment state of *v* before *expr*.
- The definite assignment state of *v* before *expr-second* is definitely assigned if the state of *v* after *expr-first* is either definitely assigned or "definitely assigned after false expression." Otherwise, it is not definitely assigned.
- The definite assignment statement of *v* after *expr* is determined by:
  - If the state of *v* after *expr-first* is definitely assigned, then the state of *v* after *expr* is definitely assigned.
  - Otherwise, if the state of *v* after *expr-second* is definitely assigned, and the state of *v* after *expr-first* is "definitely assigned after true expression," then the state of *v* after *expr* is definitely assigned.
  - Otherwise, if the state of *v* after *expr-second* is definitely assigned or "definitely assigned after false expression," then the state of *v* after *expr* is "definitely assigned after false expression."
  - Otherwise, if the state of *v* after *expr-first* is "definitely assigned after true expression," and the state of *v* after *expr-second* is "definitely assigned after true expression," then the state of *v* after *expr* is "definitely assigned after true expression."
  - Otherwise, the state of *v* after *expr* is not definitely assigned.

www.it-ebooks.info

In the example

```
class A
{
    static void G(int x, int y)
    {
        int i;
        if (x >= 0 || (i = y) >= 0)
        {
            // i not definitely assigned
        }
        else
        {
            // i definitely assigned
        }
        // i not definitely assigned
    }
}
```

the variable `i` is considered definitely assigned in one of the embedded statements of an `if` statement but not in the other. In the `if` statement in method `G`, the variable `i` is definitely assigned in the second embedded statement because execution of the expression `(i = y)` always precedes execution of this embedded statement. In contrast, the variable `i` is not definitely assigned in the first embedded statement, because `x >= 0` might have tested true, resulting in the variable `i` being unassigned.

5.3.3.26 *! Expressions*

For an expression *expr* of the form `!` *expr-operand*:

- The definite assignment state of *v* before *expr-operand* is the same as the definite assignment state of *v* before *expr*.
- The definite assignment state of *v* after *expr* is determined by:
  - If the state of *v* after *expr-operand* is definitely assigned, then the state of *v* after *expr* is definitely assigned.
  - If the state of *v* after *expr-operand* is not definitely assigned, then the state of *v* after *expr* is not definitely assigned.
  - If the state of *v* after *expr-operand* is "definitely assigned after false expression," then the state of *v* after *expr* is "definitely assigned after true expression."
  - If the state of *v* after *expr-operand* is "definitely assigned after true expression," then the state of *v* after *expr* is "definitely assigned after false expression."

www.it-ebooks.info

5.3.3.27 *?? Expressions*

For an expression *expr* of the form *expr-first* `??` *expr-second*:

- The definite assignment state of *v* before *expr-first* is the same as the definite assignment state of *v* before *expr*.
- The definite assignment state of *v* before *expr-second* is the same as the definite assignment state of *v* after *expr-first*.
- The definite assignment state of *v* after *expr* is determined by:
  - If *expr-first* is a constant expression (§7.19) with value `null`, then the state of *v* after *expr* is the same as the state of *v* after *expr-second*.
- Otherwise, the state of *v* after *expr* is the same as the definite assignment state of *v* after *expr-first*.

5.3.3.28 *?: Expressions*

For an expression *expr* of the form *expr-cond* `?` *expr-true* `:` *expr-false*:

- The definite assignment state of *v* before *expr-cond* is the same as the state of *v* before *expr*.
- The definite assignment state of *v* before *expr-true* is definitely assigned if and only if the state of *v* after *expr-cond* is definitely assigned or "definitely assigned after true expression."
- The definite assignment state of *v* before *expr-false* is definitely assigned if and only if the state of *v* after *expr-cond* is definitely assigned or "definitely assigned after false expression."
- The definite assignment state of *v* after *expr* is determined by:
  - If *expr-cond* is a constant expression (§7.19) with value `true`, then the state of *v* after *expr* is the same as the state of *v* after *expr-true*.
  - Otherwise, if *expr-cond* is a constant expression (§7.19) with value `false`, then the state of *v* after *expr* is the same as the state of *v* after *expr-false*.
  - Otherwise, if the state of *v* after *expr-true* is definitely assigned and the state of *v* after *expr-false* is definitely assigned, then the state of *v* after *expr* is definitely assigned.
  - Otherwise, the state of *v* after *expr* is not definitely assigned.

www.it-ebooks.info

5.3.3.29 *Anonymous Functions*

For a *lambda-expression* or *anonymous-method-expression expr* with a body (either *block* or *expression*) *body*:

- The definite assignment state of an outer variable *v* before *body* is the same as the state of *v* before *expr*. That is, definite assignment state of outer variables is inherited from the context of the anonymous function.
- The definite assignment state of an outer variable *v* after *expr* is the same as the state of *v* before *expr*.

The example

```
delegate bool Filter(int i);

void F()
{
    int max;

    // Error: max is not definitely assigned
    Filter f = (int n) => n < max;

    max = 5;
    DoWork(f);
}
```

generates a compile-time error because `max` is not definitely assigned where the anonymous function is declared. The example

```
delegate void D();

void F() {
    int n;
    D d = () => { n = 1; };

    d();

    // Error: n is not definitely assigned
    Console.WriteLine(n);
}
```

also generates a compile-time error because the assignment to `n` in the anonymous function has no effect on the definite assignment state of `n` outside the anonymous function.

5.4 Variable References

A *variable-reference* is an *expression* that is classified as a variable. A *variable-reference* denotes a storage location that can be accessed both to fetch the current value and to store a new value.

www.it-ebooks.info

variable-reference:
    expression

In C and C++, a *variable-reference* is known as an *lvalue*.

5.5 Atomicity of Variable References

Reads and writes of the following data types are atomic: `bool`, `char`, `byte`, `sbyte`, `short`, `ushort`, `uint`, `int`, `float`, and reference types. In addition, reads and writes of enum types with an underlying type in the previous list are also atomic. Reads and writes of other types, including `long`, `ulong`, `double`, and `decimal`, as well as user-defined types, are not guaranteed to be atomic. Aside from the library functions designed for that purpose, there is no guarantee of atomic read-modify-write, such as in the case of increment.

ERIC LIPPERT The specification authors make certain assumptions about their readers—for instance, that the reader who cares about atomicity already knows what it is. If you're not one of those readers, the issue here is that if one thread is writing the `long` 0x0123456776543210 into a variable currently holding zero, then there might be a moment in time when another thread could read 0x0123456700000000 from that variable. This assignment is not "atomic" because it could happen in two distinct phases: first the top 32 bits, then the bottom 32 bits (or vice versa). The C# language guarantees you that this never happens with 32-bit data types, but makes no guarantees about anything larger.

JON SKEET There is often confusion between atomicity and other complicated aspects of memory models. In particular, just because an assignment is atomic doesn't mean it will be necessarily be seen immediately (or, indeed, at all) by other threads, unless synchronization techniques such as volatile variables, locks, or explicit memory barriers are involved.

www.it-ebooks.info

This page intentionally left blank

www.it-ebooks.info

6. Conversions

A ***conversion*** enables an expression to be treated as being of a particular type. A conversion may cause an expression of a given type to be treated as having a different type, or it may cause an expression without a type to get a type. Conversions can be ***implicit*** or ***explicit***, and this determines whether an explicit cast is required. For instance, the conversion from type `int` to type `long` is implicit, so expressions of type `int` can implicitly be treated as type `long`. The opposite conversion, from type `long` to type `int`, is explicit and so an explicit cast is required.

```
int a = 123;
long b = a;       // Implicit conversion from int to long
int c = (int) b;  // Explicit conversion from long to int
```

Some conversions are defined by the language. Programs may also define their own conversions (§6.4).

ERIC LIPPERT The classification of conversions into "explicit" and "implicit" is handy when the developer needs to know whether a given conversion might fail at runtime. None of the implicit conversions ever fail, but explicit conversions might.

Another way of partitioning conversions that this specification does not consider in depth would be into representation-*preserving* and representation-*changing* conversions. For example, an explicit conversion from a base class (`Animal`) to a derived class (`Giraffe`) might fail at runtime if the expression converted is not an instance of `Giraffe`. If it does succeed, at least we know that the conversion will result in exactly the same object. An implicit conversion from `int` to `double`, however, always succeeds but always changes the representation; a brand-new `double` value is created, which has a completely different representation than the integer.

This partitioning of conversions will become important when we get to covariant array conversions.

www.it-ebooks.info

6.1 Implicit Conversions

The following conversions are classified as implicit conversions:

- Identity conversions.
- Implicit numeric conversions.
- Implicit enumeration conversions.
- Implicit nullable conversions.
- Null literal conversions.
- Implicit reference conversions.
- Boxing conversions.
- Implicit dynamic conversions.
- Implicit constant expression conversions.
- User-defined implicit conversions.
- Anonymous function conversions.
- Method group conversions.

Implicit conversions can occur in a variety of situations, including function member invocations (§7.5.4), cast expressions (§7.7.6), and assignments (§7.17).

The predefined implicit conversions always succeed and never cause exceptions to be thrown. Properly designed user-defined implicit conversions should exhibit these characteristics as well.

For the purposes of conversion, the types `object` and `dynamic` are considered equivalent. However, dynamic conversions (§6.1.8 and §6.2.6) apply only to expressions of type `dynamic` (§4.7).

6.1.1 Identity Conversion

An identity conversion converts from any type to the same type. This conversion exists such that an entity that already has a required type can be said to be convertible to that type.

Because `object` and `dynamic` are considered equivalent, there is an identity conversion between `object` and `dynamic`, and between constructed types that are the same when replacing all occurrences of `dynamic` with `object`.

www.it-ebooks.info

VLADIMIR RESHETNIKOV Thus identity conversion is symmetric: If type `T` has an identity conversion to type `S`, then type `S` also has an identity conversion to type `T`. If two types have an identity conversion between them, they always have identical runtime representations. For example, `List<object[]>` and `List<dynamic[]>` both are represented as `List<object[]>` at runtime.

BILL WAGNER This is an important property of the `dynamic` type. It means types such as `List<object>` and `List<dynamic>` have an identity conversion. You can convert a collection of `object` types to a collection of `dynamic` types. Furthermore, this identity relationship does not exist between `dynamic` and any type derived from `object`. That is, `List<string>` and `List<dynamic>` do not have the identity conversion.

6.1.2 Implicit Numeric Conversions

The implicit numeric conversions are:

- From `sbyte` to `short`, `int`, `long`, `float`, `double`, or `decimal`.
- From `byte` to `short`, `ushort`, `int`, `uint`, `long`, `ulong`, `float`, `double`, or `decimal`.
- From `short` to `int`, `long`, `float`, `double`, or `decimal`.
- From `ushort` to `int`, `uint`, `long`, `ulong`, `float`, `double`, or `decimal`.
- From `int` to `long`, `float`, `double`, or `decimal`.
- From `uint` to `long`, `ulong`, `float`, `double`, or `decimal`.
- From `long` to `float`, `double`, or `decimal`.
- From `ulong` to `float`, `double`, or `decimal`.
- From `char` to `ushort`, `int`, `uint`, `long`, `ulong`, `float`, `double`, or `decimal`.
- From `float` to `double`.

Conversions from `int`, `uint`, `long`, or `ulong` to `float` and from `long` or `ulong` to `double` may cause a loss of precision, but will never cause a loss of magnitude. The other implicit numeric conversions never lose any information.

There are no implicit conversions to the `char` type, so values of the other integral types do not automatically convert to the `char` type.

www.it-ebooks.info

6.1.3 Implicit Enumeration Conversions

An implicit enumeration conversion permits the *decimal-integer-literal* `0` to be converted to any *enum-type* and to any *nullable-type* whose underlying type is an *enum-type*. In the latter case, the conversion is evaluated by converting to the underlying *enum-type* and wrapping the result (§4.1.10).

ERIC LIPPERT This issue is particularly important for enums that represent a set of flags. To be compliant with CLR guidelines, it is a good idea for enum types intended to be used as flags to be marked with the `[Flags]` attribute and to have a "None" value set to zero. But for those that do not meet these criteria, it is nice to be able to assign zero without having to cast.

Lots more dos and don'ts exist regarding proper use of enumerated types. See Section 4.8 of "Framework Design Guidelines" for details.

The Microsoft implementation of C# allows *any* constant zero to go to any enum, not just *literal* constant zeros. This minor specification violation exists for historical reasons.

JOSEPH ALBAHARI The default value for an enum type is `0`, so assigning the literal `0` provides a consistent means of resetting an enum variable to its default value. For combinable flags-based enums, `0` means "no flags."

6.1.4 Implicit Nullable Conversions

Predefined implicit conversions that operate on non-nullable value types can also be used with nullable forms of those types. For each of the predefined implicit identity and numeric conversions that convert from a non-nullable value type `S` to a non-nullable value type `T`, the following implicit nullable conversions exist:

- An implicit conversion from `S?` to `T?`.
- An implicit conversion from `S` to `T?`.

Evaluation of an implicit nullable conversion based on an underlying conversion from `S` to `T` proceeds as follows:

- If the nullable conversion is from `S?` to `T?`:
  - If the source value is null (`HasValue` property is `false`), the result is the null value of type `T?`.

www.it-ebooks.info

  - Otherwise, the conversion is evaluated as an unwrapping from `S?` to `S`, followed by the underlying conversion from `S` to `T`, followed by a wrapping (§4.1.10) from `T` to `T?`.
- If the nullable conversion is from `S` to `T?`, the conversion is evaluated as the underlying conversion from `S` to `T`, followed by a wrapping from `T` to `T?`.

6.1.5 Null Literal Conversions

An implicit conversion exists from the `null` literal to any nullable type. This conversion produces the null value (§4.1.10) of the given nullable type.

6.1.6 Implicit Reference Conversions

The implicit reference conversions are:

- From any *reference-type* to `object` and `dynamic`.
- From any *class-type* `S` to any *class-type* `T`, provided `S` is derived from `T`.
- From any *class-type* `S` to any *interface-type* `T`, provided `S` implements `T`.
- From any *interface-type* `S` to any *interface-type* `T`, provided `S` is derived from `T`.
- From an *array-type* `S` with an element type S_E to an *array-type* `T` with an element type T_E, provided all of the following are true:
  - `S` and `T` differ only in element type. In other words, `S` and `T` have the same number of dimensions.
  - Both S_E and T_E are *reference-type*s.
  - An implicit reference conversion exists from S_E to T_E.

BILL WAGNER The fact that both `SE` and `TE` must be *reference-type*s means that array conversion is illegal for arrays of numeric types.

VLADIMIR RESHETNIKOV Actually, the Microsoft C# compiler does not check that `SE` and `TE` are *reference-type*s; it checks only that the implicit conversion from `SE` to `TE` is classified as an implicit **reference** conversion. This distinction is quite subtle, but it allows implicit conversions between arrays of type parameters. Note that in the following example, the type parameter `T` is not even formally *known to be a reference type* (§10.1.5), although it always is a reference type at runtime:

```
T[] Foo<T,S>(S[] x) where S: class, T
{
    return x;
}
```


www.it-ebooks.info

ERIC LIPPERT Covariant array conversions were a controversial addition to the CLI and C#. This kind of conversion breaks type safety on arrays. You might assume that you can put a `Turtle` into an array of `Animals`, but it is legal to implicitly convert an array of `Giraffes` to an expression typed as being an array of `Animals`. When you attempt to add the `Turtle` to that thing, the runtime environment will disallow the conversion from `Turtle` to `Giraffe`. Thus it is sometimes *not* legal to put a `Turtle` into an array of `Animals`, and you will not know that fact until runtime. The compiler is unable to catch this type of error.

On such occasions, it would be useful to characterize conversions as representation-*preserving* or representation-*changing*, rather than implicit or explicit. The CLI rules for covariant array conversions require that the conversion from the source to the destination type preserve representation at runtime. For example, converting an array of 10 integers into an array of 10 doubles would end up allocating memory for each double, so it could not be done cheaply "in place." By contrast, converting an array of `Giraffes` into an array of `Animals` preserves the representation of every element in the array at runtime, thereby preserving the representation of the array itself.

Because reference conversions always preserve the representation, the C# language restricts explicit and implicit covariant array conversions to those where the conversion between the element types is a reference conversion. The CLR additionally allows other representation-preserving conversions such as `int[]` to `uint[]`.

- From any *array-type* to `System.Array` and the interfaces it implements.
- From a single-dimensional array type `S[]` to `System.Collections.Generic.IList<T>` and its base interfaces, provided that there is an implicit identity or reference conversion from `S` to `T`.
- From any *delegate-type* to `System.Delegate` and the interfaces it implements.
- From the null literal to any *reference-type*.
- From any *reference-type* to a *reference-type* `T` if it has an implicit identity or reference conversion to a *reference-type* T_0 and T_0 has an identity conversion to `T`.

VLADIMIR RESHETNIKOV For example, there is an implicit reference conversion from `List<object>` to `ICollection<dynamic>`.

- From any *reference-type* to an interface or delegate type `T` if it has an implicit identity or reference conversion to an interface or delegate type T_0 and T_0 is variance-convertible (§13.1.3.2) to `T`.

www.it-ebooks.info

VLADIMIR RESHETNIKOV For example, there is an implicit reference conversion from `List<string>` to `IEnumerable<object>`.

- Implicit conversions involving type parameters that are known to be reference types. See §6.1.10 for more details on implicit conversions involving type parameters.

The implicit reference conversions are those conversions between *reference-type*s that can be proven to always succeed and, therefore, require no checks at runtime.

Reference conversions, implicit or explicit, never change the referential identity of the object being converted. In other words, while a reference conversion may change the type of the reference, it never changes the type or value of the object being referred to.

6.1.7 Boxing Conversions

A boxing conversion permits a *value-type* to be implicitly converted to a reference type. A boxing conversion exists from any *non-nullable-value-type* to `object` and `dynamic`, to `System.ValueType` and to any *interface-type* implemented by the *non-nullable-value-type*. Furthermore an *enum-type* can be converted to the type `System.Enum`.

A boxing conversion exists from a *nullable-type* to a reference type, if and only if a boxing conversion exists from the underlying *non-nullable-value-type* to the reference type.

A value type has a boxing conversion to an interface type `I` if it has a boxing conversion to an interface type $\texttt{I}_0$ and $\texttt{I}_0$ has an identity conversion to `I`.

A value type has a boxing conversion to an interface type `I` if it has a boxing conversion to an interface or delegate type $\texttt{I}_0$ and $\texttt{I}_0$ is variance-convertible (§13.1.3.2) to `I`.

Boxing a value of a *non-nullable-value-type* consists of allocating an object instance and copying the *value-type* value into that instance. A struct can be boxed to the type `System.ValueType`, since that is a base class for all structs (§11.3.2).

Boxing a value of a *nullable-type* proceeds as follows:

- If the source value is null (`HasValue` property is `false`), the result is a null reference of the target type.
- Otherwise, the result is a reference to a boxed `T` produced by unwrapping and boxing the source value.

Boxing conversions are described further in §4.3.1.

www.it-ebooks.info

6.1.8 Implicit Dynamic Conversions

An implicit dynamic conversion exists from an expression of type `dynamic` to any type `T`. The conversion is dynamically bound (§7.2.2), which means that an implicit conversion will be sought at runtime from the runtime type of the expression to `T`. If no conversion is found, a runtime exception is thrown.

VLADIMIR RESHETNIKOV This does not imply that an implicit conversion from **type** `dynamic` to any type `T` exists—which is an important difference in some overload resolution scenarios:

```
class A
{
    static void Foo(string x) { }
    static void Foo(dynamic x) { }

    static void Main()
    {
        Foo(null);
    }
}
```

The overloaded `Foo(string x)` is better than `Foo(dynamic x)`, because there is an implicit conversion from `string` to `dynamic`, but *not* from `dynamic` to `string`.

Note that this implicit conversion seemingly violates the advice in the beginning of §6.1 that an implicit conversion should never cause an exception. However, it is not the conversion itself, but the *finding* of the conversion, that causes the exception. The risk of runtime exceptions is inherent in the use of dynamic binding. If dynamic binding of the conversion is not desired, the expression can be first converted to `object`, and then to the desired type.

The following example illustrates implicit dynamic conversions:

```
object o  = "object"
dynamic d = "dynamic";

string s1 = o; // Fails at compile time: no conversion exists
string s2 = d; // Compiles and succeeds at runtime
int i     = d; // Compiles but fails at runtime: no conversion exists
```

The assignments to `s2` and `i` both employ implicit dynamic conversions, where the binding of the operations is suspended until runtime. At runtime, implicit conversions are sought from the runtime type of `d` – `string` – to the target type. A conversion is found to `string` but not to `int`.

www.it-ebooks.info

6.1.9 Implicit Constant Expression Conversions

An implicit constant expression conversion permits the following conversions:

- A *constant-expression* (§7.19) of type `int` can be converted to type `sbyte`, `byte`, `short`, `ushort`, `uint`, or `ulong`, provided the value of the *constant-expression* is within the range of the destination type.

> **JON SKEET** C#'s approach here is more flexible than Java's: Java permits implicit constant expression conversions only as assignment conversions. For example, the following code is legal in C#, but the equivalent Java code would fail to compile because the integer expression 10 would not be converted to a `byte`:
>
> ```
> void MethodTakingByte(byte b) { ... }
> ...
> // In another method
> MethodTakingByte(10);
> ```

- A *constant-expression* of type `long` can be converted to type `ulong`, provided the value of the *constant-expression* is not negative.

6.1.10 Implicit Conversions Involving Type Parameters

The following implicit conversions exist for a given type parameter `T`:

- From `T` to its effective base class `C`, from `T` to any base class of `C`, and from `T` to any interface implemented by `C`. At runtime, if `T` is a value type, the conversion is executed as a boxing conversion. Otherwise, the conversion is executed as an implicit reference conversion or identity conversion.
- From `T` to an interface type `I` in `T`'s effective interface set and from `T` to any base interface of `I`. At runtime, if `T` is a value type, the conversion is executed as a boxing conversion. Otherwise, the conversion is executed as an implicit reference conversion or identity conversion.
- From `T` to a type parameter `U`, provided `T` depends on `U` (§10.1.5). At runtime, if `U` is a value type, then `T` and `U` are necessarily the same type and no conversion is performed. Otherwise, if `T` is a value type, the conversion is executed as a boxing conversion. Otherwise, the conversion is executed as an implicit reference conversion or identity conversion.
- From the null literal to `T`, provided `T` is known to be a reference type.
- From `T` to a reference type `I` if it has an implicit conversion to a reference type S_0 and S_0 has an identity conversion to `S`. At runtime, the conversion is executed the same way as the conversion to S_0.

www.it-ebooks.info

- From `T` to an interface type `I`, if it has an implicit conversion to an interface or delegate type I_0 and I_0 is variance-convertible to `I` (§13.1.3.2). At runtime, if `T` is a value type, the conversion is executed as a boxing conversion. Otherwise, the conversion is executed as an implicit reference conversion or identity conversion.

If `T` is known to be a reference type (§10.1.5), the conversions above are all classified as implicit reference conversions (§6.1.6). If `T` is *not* known to be a reference type, the conversions above are classified as boxing conversions (§6.1.7).

6.1.11 User-Defined Implicit Conversions

A user-defined implicit conversion consists of an optional standard implicit conversion, followed by execution of a user-defined implicit conversion operator, followed by another optional standard implicit conversion. The exact rules for evaluating user-defined implicit conversions are described in §6.4.4.

6.1.12 Anonymous Function Conversions and Method Group Conversions

Anonymous functions and method groups do not have types in and of themselves, but may be implicitly converted to delegate types or expression tree types. Anonymous function conversions are described in more detail in §6.5 and method group conversions in §6.6.

6.2 Explicit Conversions

The following conversions are classified as explicit conversions:

- All implicit conversions.
- Explicit numeric conversions.
- Explicit enumeration conversions.
- Explicit nullable conversions.
- Explicit reference conversions.
- Explicit interface conversions.
- Unboxing conversions.
- Explicit dynamic conversions.
- User-defined explicit conversions.

Explicit conversions can occur in cast expressions (§7.7.6).

The set of explicit conversions includes all implicit conversions. This means that redundant cast expressions are allowed.

www.it-ebooks.info

The explicit conversions that are not implicit conversions are conversions that cannot be proven to always succeed, conversions that are known to possibly lose information, and conversions across domains of types sufficiently different to merit explicit notation.

6.2.1 Explicit Numeric Conversions

The explicit numeric conversions are the conversions from a *numeric-type* to another *numeric-type* for which an implicit numeric conversion (§6.1.2) does not already exist:

- From `sbyte` to `byte`, `ushort`, `uint`, `ulong`, or `char`.
- From `byte` to `sbyte` and `char`.
- From `short` to `sbyte`, `byte`, `ushort`, `uint`, `ulong`, or `char`.
- From `ushort` to `sbyte`, `byte`, `short`, or `char`.
- From `int` to `sbyte`, `byte`, `short`, `ushort`, `uint`, `ulong`, or `char`.
- From `uint` to `sbyte`, `byte`, `short`, `ushort`, `int`, or `char`.
- From `long` to `sbyte`, `byte`, `short`, `ushort`, `int`, `uint`, `ulong`, or `char`.
- From `ulong` to `sbyte`, `byte`, `short`, `ushort`, `int`, `uint`, `long`, or `char`.
- From `char` to `sbyte`, `byte`, or `short`.
- From `float` to `sbyte`, `byte`, `short`, `ushort`, `int`, `uint`, `long`, `ulong`, `char`, or `decimal`.
- From `double` to `sbyte`, `byte`, `short`, `ushort`, `int`, `uint`, `long`, `ulong`, `char`, `float`, or `decimal`.
- From `decimal` to `sbyte`, `byte`, `short`, `ushort`, `int`, `uint`, `long`, `ulong`, `char`, `float`, or `double`.

Because the explicit conversions include all implicit and explicit numeric conversions, it is always possible to convert from any *numeric-type* to any other *numeric-type* using a cast expression (§7.7.6).

The explicit numeric conversions possibly lose information or possibly cause exceptions to be thrown. An explicit numeric conversion is processed as follows:

- For a conversion from an integral type to another integral type, the processing depends on the overflow checking context (§7.6.12) in which the conversion takes place:
  - In a `checked` context, the conversion succeeds if the value of the source operand is within the range of the destination type, but throws a `System.OverflowException` if the value of the source operand is outside the range of the destination type.
  - In an `unchecked` context, the conversion always succeeds, and proceeds as follows:

www.it-ebooks.info

- If the source type is larger than the destination type, then the source value is truncated by discarding its "extra" most significant bits. The result is then treated as a value of the destination type.
- If the source type is smaller than the destination type, then the source value is either sign-extended or zero-extended so that it is the same size as the destination type. Sign-extension is used if the source type is signed; zero-extension is used if the source type is unsigned. The result is then treated as a value of the destination type.
- If the source type is the same size as the destination type, then the source value is treated as a value of the destination type.

- For a conversion from `decimal` to an integral type, the source value is rounded toward zero to the nearest integral value, and this integral value becomes the result of the conversion. If the resulting integral value is outside the range of the destination type, a `System.OverflowException` is thrown.
- For a conversion from `float` or `double` to an integral type, the processing depends on the overflow checking context (§7.6.12) in which the conversion takes place:
  - In a `checked` context, the conversion proceeds as follows:
    - If the value of the operand is NaN or infinite, a `System.OverflowException` is thrown.
    - Otherwise, the source operand is rounded toward zero to the nearest integral value. If this integral value is within the range of the destination type, then this value is the result of the conversion.
    - Otherwise, a `System.OverflowException` is thrown.
  - In an `unchecked` context, the conversion always succeeds, and proceeds as follows:
    - If the value of the operand is NaN or infinite, the result of the conversion is an unspecified value of the destination type.
    - Otherwise, the source operand is rounded toward zero to the nearest integral value. If this integral value is within the range of the destination type, then this value is the result of the conversion.
    - Otherwise, the result of the conversion is an unspecified value of the destination type.
- For a conversion from `double` to `float`, the `double` value is rounded to the nearest `float` value. If the `double` value is too small to represent as a `float`, the result becomes positive zero or negative zero. If the `double` value is too large to represent as a `float`, the result becomes positive infinity or negative infinity. If the `double` value is NaN, the result is also NaN.

www.it-ebooks.info

- For a conversion from `float` or `double` to `decimal`, the source value is converted to `decimal` representation and rounded to the nearest number after the 28th decimal place if required (§4.1.7). If the source value is too small to represent as a `decimal`, the result becomes zero. If the source value is NaN, infinity, or too large to represent as a `decimal`, a `System.OverflowException` is thrown.
- For a conversion from `decimal` to `float` or `double`, the `decimal` value is rounded to the nearest `double` or `float` value. While this conversion may lose precision, it never causes an exception to be thrown.

JOSEPH ALBAHARI The round-to-zero behavior in the real-to-integral conversions is efficient but not always the most useful approach. For instance, under this scheme, `(int) 3.9` evaluates to `3`, rather than `4`. C# provides no built-in mechanism for converting to the nearest integer; this capability is left to external libraries. In Microsoft's .NET Framework, the static `Convert` class provides this functionality via methods such as `ToInt32`.

6.2.2 Explicit Enumeration Conversions

The explicit enumeration conversions are:

- From `sbyte`, `byte`, `short`, `ushort`, `int`, `uint`, `long`, `ulong`, `char`, `float`, `double`, or `decimal` to any *enum-type*.
- From any *enum-type* to `sbyte`, `byte`, `short`, `ushort`, `int`, `uint`, `long`, `ulong`, `char`, `float`, `double`, or `decimal`.
- From any *enum-type* to any other *enum-type*.

An explicit enumeration conversion between two types is processed by treating any participating *enum-type* as the underlying type of that *enum-type*, and then performing an implicit or explicit numeric conversion between the resulting types. For example, given an *enum-type* `E` with an underlying type of `int`, a conversion from `E` to `byte` is processed as an explicit numeric conversion (§6.2.1) from `int` to `byte`, and a conversion from `byte` to `E` is processed as an implicit numeric conversion (§6.1.2) from `byte` to `int`.

6.2.3 Explicit Nullable Conversions

Explicit nullable conversions permit predefined explicit conversions that operate on non-nullable value types to also be used with nullable forms of those types. For each of the predefined explicit conversions that convert from a non-nullable value type `S` to a non-nullable value type `T` (§6.1.1, §6.1.2, §6.1.3, §6.2.1, and §6.2.2), the following nullable conversions exist:

www.it-ebooks.info

- An explicit conversion from `S?` to `T?`.
- An explicit conversion from `S` to `T?`.
- An explicit conversion from `S?` to `T`.

Evaluation of a nullable conversion based on an underlying conversion from `S` to `T` proceeds as follows:

- If the nullable conversion is from `S?` to `T?`:
  - If the source value is null (`HasValue` property is `false`), the result is the null value of type `T?`.
  - Otherwise, the conversion is evaluated as an unwrapping from `S?` to `S`, followed by the underlying conversion from `S` to `T`, followed by a wrapping from `T` to `T?`.
- If the nullable conversion is from `S` to `T?`, the conversion is evaluated as the underlying conversion from `S` to `T` followed by a wrapping from `T` to `T?`.
- If the nullable conversion is from `S?` to `T`, the conversion is evaluated as an unwrapping from `S?` to `S` followed by the underlying conversion from `S` to `T`.

Note that an attempt to unwrap a nullable value will throw an exception if the value is `null`.

6.2.4 Explicit Reference Conversions

The explicit reference conversions are:

- From `object` and `dynamic` to any other *reference-type*.
- From any *class-type* `S` to any *class-type* `T`, provided `S` is a base class of `T`.
- From any *class-type* `S` to any *interface-type* `T`, provided `S` is not sealed and provided `S` does not implement `T`.
- From any *interface-type* `S` to any *class-type* `T`, provided `T` is not sealed or provided `T` implements `S`.
- From any *interface-type* `S` to any *interface-type* `T`, provided `S` is not derived from `T`.
- From an *array-type* `S` with an element type S_E to an *array-type* `T` with an element type T_E, provided all of the following are true:
  - `S` and `T` differ only in element type. In other words, `S` and `T` have the same number of dimensions.
  - Both S_E and T_E are *reference-type*s.
  - An explicit reference conversion exists from S_E to T_E.

www.it-ebooks.info

VLADIMIR RESHETNIKOV The Microsoft C# compiler does not check whether SE and TE are *reference-types*; it only checks whether the explicit conversion from SE to TE is classified as an explicit **reference** conversion. See the corresponding annotation in §6.1.6 for details.

- From System.Array and the interfaces it implements to any *array-type*.
- From a single-dimensional array type S[] to System.Collections.Generic.IList<T> and its base interfaces, provided that there is an explicit reference conversion from S to T.
- From System.Collections.Generic.IList<S> and its base interfaces to a single-dimensional array type T[], provided that there is an explicit identity or reference conversion from S to T.
- From System.Delegate and the interfaces it implements to any *delegate-type*.
- From a reference type to a reference type T, if it has an explicit reference conversion to a reference type T_0 and T_0 has an identity conversion T.
- From a reference type to an interface or delegate type T, if it has an explicit reference conversion to an interface or delegate type T_0 and either T_0 is variance-convertible to T or T is variance-convertible to T_0 (§13.1.3.2).
- From $D<S_1...S_n>$ to a $D<T_1...T_n>$, where $D<X_1...X_n>$ is a generic delegate type, $D<S_1...S_n>$ is not compatible with or identical to $D<T_1...T_n>$, and for each type parameter X_i of D the following holds:
  - If X_i is invariant, then S_i is identical to T_i.
  - If X_i is covariant, then there is an implicit or explicit identity or reference conversion from S_i to T_i.
  - If X_i is contravariant, then S_i and T_i are either identical or both reference types.

ERIC LIPPERT The justification of this last point is that you might have a variable of type Action<S>, where S is a reference type that contains an instance of Action<object>. You can *implicitly* convert an Action<object> to Action<T>, where T is any reference type, so you should be able to *explicitly* convert Action<S> to Action<T> for any reference types S and T.

- Explicit conversions involving type parameters that are known to be reference types. For more details on explicit conversions involving type parameters, see §6.2.6.

The explicit reference conversions are those conversions between reference types that require runtime checks to ensure they are correct.

www.it-ebooks.info

For an explicit reference conversion to succeed at runtime, the value of the source operand must be null, or the *actual* type of the object referenced by the source operand must be a type that can be converted to the destination type by an implicit reference conversion (§6.1.6) or boxing conversion (§6.1.7). If an explicit reference conversion fails, a `System.InvalidCastException` is thrown.

Reference conversions, implicit or explicit, never change the referential identity of the object being converted. In other words, while a reference conversion may change the type of the reference, it never changes the type or value of the object being referred to.

6.2.5 Unboxing Conversions

An unboxing conversion permits a reference type to be explicitly converted to a *value-type*. An unboxing conversion exists from the types `object`, `dynamic`, and `System.ValueType` to any *non-nullable-value-type*, and from any *interface-type* to any *non-nullable-value-type* that implements the *interface-type*. Furthermore, type `System.Enum` can be unboxed to any *enum-type*.

An unboxing conversion exists from a reference type to a *nullable-type* if an unboxing conversion exists from the reference type to the underlying *non-nullable-value-type* of the *nullable-type*.

A value type `S` has an unboxing conversion from an interface type `I` if it has an unboxing conversion from an interface type $\texttt{I}_0$ and $\texttt{I}_0$ has an identity conversion to `I`.

A value type `S` has an unboxing conversion from an interface type `I` if it has an unboxing conversion from an interface or delegate type $\texttt{I}_0$ and either $\texttt{I}_0$ is variance-convertible to `I` or `I` is variance-convertible to $\texttt{I}_0$ (§13.1.3.2).

An unboxing operation consists of first checking that the object instance is a boxed value of the given *value-type*, and then copying the value out of the instance. Unboxing a null reference to a *nullable-type* produces the null value of the *nullable-type*. A struct can be unboxed from the type `System.ValueType`, since that is a base class for all structs (§11.3.2).

Unboxing conversions are described further in §4.3.2.

6.2.6 Explicit Dynamic Conversions

An explicit dynamic conversion exists from an expression of type `dynamic` to any type `T`. The conversion is dynamically bound (§7.2.2), which means that an explicit conversion will be sought at runtime from the runtime type of the expression to `T`. If no conversion is found, a runtime exception is thrown.

If dynamic binding of the conversion is not desired, the expression can be first converted to `object`, and then to the desired type.

www.it-ebooks.info

Assume the following class is defined:

```
class C
{
    int i;

    public C(int i) { this.i = i; }

    public static explicit operator C(string s)
    {
        return new C(int.Parse(s));
    }
}
```

The following example illustrates explicit dynamic conversions:

```
object o  = "1";
dynamic d = "2";

var c1 = (C)o; // Compiles, but explicit reference conversion fails
var c2 = (C)d; // Compiles and user-defined conversion succeeds
```

The best conversion of `o` to `C` is found at compile time to be an explicit reference conversion. This fails at runtime, because `"1"` is not, in fact, a `C`. The conversion of `d` to `C` as an explicit dynamic conversion, however, is suspended to runtime, where a user-defined conversion from the runtime type of `d` – `string` – to `C` is found, and succeeds.

6.2.7 Explicit Conversions Involving Type Parameters

The following explicit conversions exist for a given type parameter `T`:

- From the effective base class `C` of `T` to `T` and from any base class of `C` to `T`. At runtime, if `T` is a value type, the conversion is executed as an unboxing conversion. Otherwise, the conversion is executed as an explicit reference conversion or identity conversion.
- From any interface type to `T`. At runtime, if `T` is a value type, the conversion is executed as an unboxing conversion. Otherwise, the conversion is executed as an explicit reference conversion or identity conversion.
- From `T` to any *interface-type* `I`, provided there is not already an implicit conversion from `T` to `I`. At runtime, if `T` is a value type, the conversion is executed as a boxing conversion followed by an explicit reference conversion. Otherwise, the conversion is executed as an explicit reference conversion or identity conversion.
- From a type parameter `U` to `T`, provided `T` depends on `U` (§10.1.5). At runtime, if `U` is a value type, then `T` and `U` are necessarily the same type and no conversion is performed. Otherwise, if `T` is a value type, the conversion is executed as an unboxing conversion. Otherwise, the conversion is executed as an explicit reference conversion or identity conversion.

www.it-ebooks.info

If T is known to be a reference type, all of these conversions are classified as explicit reference conversions (§6.2.4). If T is *not* known to be a reference type, these conversions are classified as unboxing conversions (§6.2.5).

JOSEPH ALBAHARI The behavior of conversions involving type parameters can be further illustrated with the following method:

```
static T Cast<T>(object value) { return (T)value; }
```

This method can perform a reference conversion or an unboxing, but never a numeric or user-defined conversion:

```
string s = Cast<string>("s");
         // Okay - reference conversion
int i    = Cast<int>(3);
         // Okay - unboxing
long l   = Cast<long>(3);
         // InvalidCastException: attempts
         // an unboxing instead of an
         // int->long numeric conversion
```

An identical situation arises in the method `System.Linq.Enumerable.Cast`, which is a standard query operator in Microsoft's LINQ implementation.

The above rules do not permit a direct explicit conversion from an unconstrained type parameter to a non-interface type, which might be surprising. The reason for this rule is to prevent confusion and make the semantics of such conversions clear. For example, consider the following declaration:

```
class X<T>
{
    public static long F(T t) {
        return (long)t;   // Error
    }
}
```

If the direct explicit conversion of `t` to `int` were permitted, one might easily expect that `X<int>.F(7)` would return `7L`. However, it would not, because the standard numeric conversions are considered only when the types are known to be numeric at binding time. To make the semantics clear, the above example must instead be written:

```
class X<T>
{
    public static long F(T t) {
        return (long)(object)t; // Okay, but works only when T is long
    }
}
```

This code will now compile but executing `X<int>.F(7)` would then throw an exception at runtime, because a boxed `int` cannot be converted directly to a `long`.

www.it-ebooks.info

6.2.8 User-Defined Explicit Conversions

A user-defined explicit conversion consists of an optional standard explicit conversion, followed by execution of a user-defined implicit or explicit conversion operator, followed by another optional standard explicit conversion. The exact rules for evaluating user-defined explicit conversions are described in §6.4.5.

ERIC LIPPERT The conversions that "sandwich" the user-defined conversion are explicit conversions and, therefore, might fail themselves. Thus a user-defined explicit conversion has three potential points of failure at runtime, not just one.

Even so, at least the sandwiching conversions are never user-defined conversions themselves. For example, if there is a user-defined explicit conversion from `X` to `Y`, and a *user-defined* explicit conversion from `Y` to `Z`, then an attempt to cast `X` to `Z` will not succeed.

That said, the chain of explicit conversions can get quite long in contrived scenarios. For example, consider a `struct Foo` with a user-defined explicit conversion from `Foo?` to `decimal`. A cast on an expression of type `Foo` to type `int?` will convert `Foo` to `Foo?`, then `Foo?` to `decimal`, then `decimal` to `int`, and then `int` to `int?`.

6.3 Standard Conversions

The standard conversions are those predefined conversions that can occur as part of a user-defined conversion.

6.3.1 Standard Implicit Conversions

The following implicit conversions are classified as standard implicit conversions:

- Identity conversions (§6.1.1).
- Implicit numeric conversions (§6.1.2).
- Implicit nullable conversions (§6.1.4).
- Implicit reference conversions (§6.1.6).
- Boxing conversions (§6.1.7).
- Implicit constant expression conversions (§6.1.8).
- Implicit conversions involving type parameters (§6.1.10).

The standard implicit conversions specifically exclude user-defined implicit conversions.

www.it-ebooks.info

6.3.2 Standard Explicit Conversions

The standard explicit conversions are all standard implicit conversions plus the subset of the explicit conversions for which an opposite standard implicit conversion exists. In other words, if a standard implicit conversion exists from a type `A` to a type `B`, then a standard explicit conversion exists from type `A` to type `B` and from type `B` to type `A`.

> **VLADIMIR RESHETNIKOV** For example, the predefined explicit numeric conversions from `double` to `decimal` and from `decimal` to `double` are not standard explicit conversions.

6.4 User-Defined Conversions

C# allows the predefined implicit and explicit conversions to be augmented by ***user-defined conversions***. User-defined conversions are introduced by declaring conversion operators (§10.10.3) in class and struct types.

> **BILL WAGNER** Conversions, in general, imply that multiple types are somehow interchangeable. That notion, in turn, implies that one of more of the types may not be necessary. As you write more conversion methods, consider whether you might have created multiple types that serve the same purpose.
>
> When you write conversions for reference types, be sure to preserve the property that the result of a reference conversion always refers to the same object as the source.

6.4.1 Permitted User-Defined Conversions

C# permits only certain user-defined conversions to be declared. In particular, it is not possible to redefine an already existing implicit or explicit conversion.

For a given source type `S` and target type `T`, if `S` or `T` are nullable types, let S_0 and T_0 refer to their underlying types; otherwise, S_0 and T_0 are equal to `S` and `T`, respectively. A class or struct is permitted to declare a conversion from a source type `S` to a target type `T` only if all of the following are true:

- S_0 and T_0 are different types.
- Either S_0 or T_0 is the class or struct type in which the operator declaration takes place.

www.it-ebooks.info

- Neither S_0 nor T_0 is an *interface-type*.
- Excluding user-defined conversions, a conversion does not exist from `S` to `T` or from `T` to `S`.

The restrictions that apply to user-defined conversions are discussed further in §10.10.3.

6.4.2 Lifted Conversion Operators

Given a user-defined conversion operator that converts from a non-nullable value type `S` to a non-nullable value type `T`, a ***lifted conversion operator*** exists that converts from `S?` to `T?`. This lifted conversion operator performs an unwrapping from `S?` to `S`, followed by the user-defined conversion from `S` to `T`, followed by a wrapping from `T` to `T?`, except that a null-valued `S?` converts directly to a null-valued `T?`.

A lifted conversion operator has the same implicit or explicit classification as its underlying user-defined conversion operator. The term "user-defined conversion" applies to the use of both user-defined and lifted conversion operators.

6.4.3 Evaluation of User-Defined Conversions

A user-defined conversion converts a value from its type, called the ***source type***, to another type, called the ***target type***. Evaluation of a user-defined conversion centers on finding the ***most specific*** user-defined conversion operator for the particular source and target types. This determination is broken into several steps:

- Finding the set of classes and structs from which user-defined conversion operators will be considered. This set consists of the source type and its base classes and the target type and its base classes (with the implicit assumptions that only classes and structs can declare user-defined operators, and that non-class types have no base classes). For the purposes of this step, if either the source or target type is a *nullable-type*, their underlying type is used instead.
- From that set of types, determining which user-defined and lifted conversion operators are applicable. For a conversion operator to be applicable, it must be possible to perform a standard conversion (§6.3) from the source type to the operand type of the operator, and it must be possible to perform a standard conversion from the result type of the operator to the target type.
- From the set of applicable user-defined operators, determining which operator is unambiguously the most specific. In general terms, the most specific operator is the operator whose operand type is "closest" to the source type and whose result type is "closest" to the target type. User-defined conversion operators are preferred over lifted conversion operators. The exact rules for establishing the most specific user-defined conversion operator are defined in the following sections.

www.it-ebooks.info

Once a most specific user-defined conversion operator has been identified, the actual execution of the user-defined conversion involves up to three steps:

- First, if required, performing a standard conversion from the source type to the operand type of the user-defined or lifted conversion operator.
- Next, invoking the user-defined or lifted conversion operator to perform the conversion.
- Finally, if required, performing a standard conversion from the result type of the user-defined or lifted conversion operator to the target type.

Evaluation of a user-defined conversion never involves more than one user-defined or lifted conversion operator. In other words, a conversion from type `S` to type `T` will never first execute a user-defined conversion from `S` to `X` and then execute a user-defined conversion from `X` to `T`.

Exact definitions of evaluation of user-defined implicit or explicit conversions are given in the following sections. The definitions make use of the following terms:

- If a standard implicit conversion (§6.3.1) exists from a type `A` to a type `B`, and if neither `A` nor `B` is an *interface-type*, then `A` is said to be ***encompassed by*** `B`, and `B` is said to ***encompass*** `A`.
- The ***most encompassing type*** in a set of types is the one type that encompasses all other types in the set. If no single type encompasses all other types, then the set has no most encompassing type. In more intuitive terms, the most encompassing type is the "largest" type in the set—the one type to which each of the other types can be implicitly converted.
- The ***most encompassed type*** in a set of types is the one type that is encompassed by all other types in the set. If no single type is encompassed by all other types, then the set has no most encompassed type. In more intuitive terms, the most encompassed type is the "smallest" type in the set—the one type that can be implicitly converted to each of the other types.

BILL WAGNER The more conversion operators you write, the more likely it becomes that you will introduce ambiguities among conversions. Those ambiguities will make your class more difficult to use.

www.it-ebooks.info

6.4.4 User-Defined Implicit Conversions

A user-defined implicit conversion from type S to type T is processed as follows:

- Determine the types S_0 and T_0. If S and T are nullable types, S_0 and T_0 are their underlying types; otherwise, S_0 and T_0 are equal to S and T, respectively.
- Find the set of types, D, from which user-defined conversion operators will be considered. This set consists of S_0 (if S_0 is a class or struct), the base classes of S_0 (if S_0 is a class), and T_0 (if T_0 is a class or struct).
- Find the set of applicable user-defined and lifted conversion operators, U. This set consists of the user-defined and lifted implicit conversion operators declared by the classes or structs in D that convert from a type encompassing S to a type encompassed by T. If U is empty, the conversion is undefined and a compile-time error occurs.
- Find the most specific source type, S_x, of the operators in U:
  - If any of the operators in U convert from S, then S_x is S.
  - Otherwise, S_x is the most encompassed type in the combined set of source types of the operators in U. If exactly one most encompassed type cannot be found, then the conversion is ambiguous and a compile-time error occurs.
- Find the most specific target type, T_x, of the operators in U:
  - If any of the operators in U convert to T, then T_x is T.
  - Otherwise, T_x is the most encompassing type in the combined set of target types of the operators in U. If exactly one most encompassing type cannot be found, then the conversion is ambiguous and a compile-time error occurs.
- Find the most specific conversion operator:
  - If U contains exactly one user-defined conversion operator that converts from S_x to T_x, then this is the most specific conversion operator.
  - Otherwise, if U contains exactly one lifted conversion operator that converts from S_x to T_x, then this is the most specific conversion operator.
  - Otherwise, the conversion is ambiguous and a compile-time error occurs.
- Finally, apply the conversion:
  - If S is not S_x, then a standard implicit conversion from S to S_x is performed.
  - The most specific conversion operator is invoked to convert from S_x to T_x.
  - If T_x is not T, then a standard implicit conversion from T_x to T is performed.

www.it-ebooks.info

6.4.5 User-Defined Explicit Conversions

A user-defined explicit conversion from type S to type T is processed as follows:

- Determine the types S_0 and T_0. If S or T is a nullable type, S_0 and T_0 are their respective underlying types; otherwise, S_0 and T_0 are equal to S and T, respectively.
- Find the set of types, D, from which user-defined conversion operators will be considered. This set consists of S_0 (if S_0 is a class or struct), the base classes of S_0 (if S_0 is a class), T_0 (if T_0 is a class or struct), and the base classes of T_0 (if T_0 is a class).
- Find the set of applicable user-defined and lifted conversion operators, U. This set consists of the user-defined and lifted implicit or explicit conversion operators declared by the classes or structs in D that convert from a type encompassing or encompassed by S to a type encompassing or encompassed by T. If U is empty, the conversion is undefined and a compile-time error occurs.
- Find the most specific source type, S_X, of the operators in U:
  - If any of the operators in U convert from S, then S_X is S.
  - Otherwise, if any of the operators in U convert from types that encompass S, then S_X is the most encompassed type in the combined set of source types of those operators. If no most encompassed type can be found, then the conversion is ambiguous and a compile-time error occurs.
  - Otherwise, S_X is the most encompassing type in the combined set of source types of the operators in U. If exactly one most encompassing type cannot be found, then the conversion is ambiguous and a compile-time error occurs.
- Find the most specific target type, T_X, of the operators in U:
  - If any of the operators in U convert to T, then T_X is T.
  - Otherwise, if any of the operators in U convert to types that are encompassed by T, then T_X is the most encompassing type in the combined set of target types of those operators. If exactly one most encompassing type cannot be found, then the conversion is ambiguous and a compile-time error occurs.
  - Otherwise, T_X is the most encompassed type in the combined set of target types of the operators in U. If no most encompassed type can be found, then the conversion is ambiguous and a compile-time error occurs.
- Find the most specific conversion operator:
  - If U contains exactly one user-defined conversion operator that converts from S_X to T_X, then this is the most specific conversion operator.
  - Otherwise, if U contains exactly one lifted conversion operator that converts from S_X to T_X, then this is the most specific conversion operator.

www.it-ebooks.info

  - Otherwise, the conversion is ambiguous and a compile-time error occurs.
- Finally, apply the conversion:
  - If S is not S_x, then a standard explicit conversion from S to S_x is performed.
  - The most specific user-defined conversion operator is invoked to convert from S_x to T_x.
  - If T_x is not T, then a standard explicit conversion from T_x to T is performed.

6.5 Anonymous Function Conversions

An *anonymous-method-expression* or *lambda-expression* is classified as an anonymous function (§7.15). The expression does not have a type but can be implicitly converted to a compatible delegate type or expression tree type. Specifically, a delegate type D is compatible with an anonymous function F provided:

- If F contains an *anonymous-function-signature*, then D and F have the same number of parameters.
- If F does not contain an *anonymous-function-signature*, then D may have zero or more parameters of any type, as long as no parameter of D has the `out` parameter modifier.
- If F has an explicitly typed parameter list, each parameter in D has the same type and modifiers as the corresponding parameter in F.

> **VLADIMIR RESHETNIKOV** One exception to this rule: If the delegate type D has a parameter with the `params` modifier, then the corresponding parameter in F is allowed (and required) not to have any modifiers.

- If F has an implicitly typed parameter list, D has no `ref` or `out` parameters.
- If D has a `void` return type and the body of F is an expression, when each parameter of F is given the type of the corresponding parameter in D, the body of F is a valid expression (with respect to §7) that would be permitted as a *statement-expression* (§8.6).
- If D has a `void` return type and the body of F is a statement block, when each parameter of F is given the type of the corresponding parameter in D, the body of F is a valid statement block (with respect to §8.2) in which no `return` statement specifies an expression.
- If D has a non-void return type and the body of F is an expression, when each parameter of F is given the type of the corresponding parameter in D, the body of F is a valid expression (with respect to §7) that is implicitly convertible to the return type of D.
- If D has a non-void return type and the body of F is a statement block, when each parameter of F is given the type of the corresponding parameter in D, the body of F is a

www.it-ebooks.info

valid statement block (with respect to §8.2) with a non-reachable end point in which each `return` statement specifies an expression that is implicitly convertible to the return type of `D`.

An expression tree type `Expression<D>` is compatible with an anonymous function `F` if the delegate type `D` is compatible with `F`.

> **VLADIMIR RESHETNIKOV** Only anonymous functions of the form *lambda-expression* can have implicit conversions to expression tree types (*anonymous-method-expression*s are never convertible to these types). But even existing conversions from *lambda-expression*s can fail at compile time.

Certain anonymous functions cannot be converted to expression tree types: Even though the conversion *exists*, it fails at compile time. This is the case if the anonymous expression contains one or more of the following constructs:

- Simple or compound assignment operators.
- A dynamically bound expression.

> **VLADIMIR RESHETNIKOV** The other constructs not supported in the current implementation include nested *anonymous-method-expression*s, *lambda-expression*s with statement bodies, *lambda-expression*s with `ref` or `out` parameters, *base-access*, multidimensional array initializers, named and optional parameters, implicit `refs`, and pointer operations.
>
> If you need to cast an expression of type `dynamic` to another type in an expression tree, you can either use `as` operator or cast the expression to `object` first.

The examples that follow use a generic delegate type `Func<A,R>`, which represents a function that takes an argument of type `A` and returns a value of type `R`:

```
delegate R Func<A,R>(A arg);
```

In the assignments

```
Func<int,int> f1 = x => x + 1;          // Okay
Func<int,double> f2 = x => x + 1;       // Okay
Func<double,int> f3 = x => x + 1;       // Error
```

the parameter and return types of each anonymous function are determined from the type of the variable to which the anonymous function is assigned.

www.it-ebooks.info

The first assignment successfully converts the anonymous function to the delegate type `Func<int,int>` because, when `x` is given type `int`, `x + 1` is a valid expression that is implicitly convertible to type `int`.

Likewise, the second assignment successfully converts the anonymous function to the delegate type `Func<int,double>` because the result of `x + 1` (of type `int`) is implicitly convertible to type `double`.

However, the third assignment generates a compile-time error because, when `x` is given type `double`, the result of `x + 1` (of type `double`) is not implicitly convertible to type `int`.

Anonymous functions may influence overload resolution and may participate in type inference. See §7.5 for further details.

6.5.1 Evaluation of Anonymous Function Conversions to Delegate Types

Conversion of an anonymous function to a delegate type produces a delegate instance that references the anonymous function and the (possibly empty) set of captured outer variables that are active at the time of the evaluation. When the delegate is invoked, the body of the anonymous function is executed. The code in the body is executed using the set of captured outer variables referenced by the delegate.

The invocation list of a delegate produced from an anonymous function contains a single entry. The exact target object and target method of the delegate are unspecified. In particular, it is unspecified whether the target object of the delegate is `null`, the `this` value of the enclosing function member, or some other object.

Conversions of semantically identical anonymous functions with the same (possibly empty) set of captured outer variable instances to the same delegate types are permitted (but not required) to return the same delegate instance. The term "semantically identical" is used here to mean that execution of the anonymous functions will, in all cases, produce the same effects given the same arguments. This rule permits code such as the following to be optimized.

```
delegate double Function(double x);

class Test
{
    static double[] Apply(double[] a, Function f) {
        double[] result = new double[a.Length];
        for (int i = 0; i < a.Length; i++) result[i] = f(a[i]);
        return result;
    }

    static void F(double[] a, double[] b) {
        a = Apply(a, (double x) => Math.Sin(x));
        b = Apply(b, (double y) => Math.Sin(y));
        ...
    }
}
```


www.it-ebooks.info

Since the two anonymous function delegates have the same (empty) set of captured outer variables, and since the anonymous functions are semantically identical, the compiler is permitted to have the delegates refer to the same target method. Indeed, the compiler is permitted to return the very same delegate instance from both anonymous function expressions.

JON SKEET This *potential* optimization is somewhat dangerous when one considers the prospect of combining and removing delegates—which is particularly important when it comes to events. Consider this piece of code:

```
button.Click += (sender, args) => Console.WriteLine("Clicked");
button.Click -= (sender, args) => Console.WriteLine("Clicked");
```

Assuming a simple event implementation, what will be the result? Either the button's `Click` event will have the same event handlers it had before or it will have one additional event handler. Beware of "optimizations" that change behavior significantly.

VLADIMIR RESHETNIKOV The Microsoft C# compiler does not cache delegate instances within generic methods.

6.5.2 Evaluation of Anonymous Function Conversions to Expression Tree Types

Conversion of an anonymous function to an expression tree type produces an expression tree (§4.6). More precisely, evaluation of the anonymous function conversion leads to the construction of an object structure that represents the structure of the anonymous function itself. The precise structure of the expression tree, as well as the exact process for creating it, are implementation defined.

VLADIMIR RESHETNIKOV Like a delegate, an expression tree can have a set of captured outer variables.

6.5.3 Implementation Example

This section describes a possible implementation of anonymous function conversions in terms of other C# constructs. The implementation described here is based on the same principles used by the Microsoft C# compiler, but it is by no means a mandated implementation, nor is it the only one possible. It only briefly mentions conversions to expression trees, as their exact semantics are outside the scope of this specification.

The remainder of this section gives several examples of code that contains anonymous functions with different characteristics. For each example, a corresponding translation to

www.it-ebooks.info

code that uses only other C# constructs is provided. In the examples, the identifier `D` is assumed to represent the following delegate type:

```
public delegate void D();
```

The simplest form of an anonymous function is one that captures no outer variables:

```
class Test
{
    static void F() {
        D d = () => { Console.WriteLine("test"); };
    }
}
```

This can be translated to a delegate instantiation that references a compiler-generated static method in which the code of the anonymous function is placed:

```
class Test
{
    static void F() {
        D d = new D(__Method1);
    }

    static void __Method1() {
        Console.WriteLine("test");
    }
}
```

In the following example, the anonymous function references instance members of `this`:

```
class Test
{
    int x;

    void F() {
        D d = () => { Console.WriteLine(x); };
    }
}
```

This can be translated to a compiler-generated instance method containing the code of the anonymous function:

```
class Test
{
    int x;

    void F() {
        D d = new D(__Method1);
    }

    void __Method1() {
        Console.WriteLine(x);
    }
}
```


www.it-ebooks.info

In this example, the anonymous function captures a local variable:

```
class Test
{
    void F() {
        int y = 123;
        D d = () => { Console.WriteLine(y); };
    }
}
```

The lifetime of the local variable must now be extended to at least the lifetime of the anonymous function delegate. This can be achieved by "hoisting" the local variable into a field of a compiler-generated class. Instantiation of the local variable (§7.15.5.2) then corresponds to creating an instance of the compiler generated class, and accessing the local variable corresponds to accessing a field in the instance of the compiler-generated class. Furthermore, the anonymous function becomes an instance method of the compiler-generated class:

```
class Test
{
    void F() {
        __Locals1 __locals1 = new __Locals1();
        __locals1.y = 123;
        D d = new D(__locals1.__Method1);
    }

    class __Locals1
    {
        public int y;

        public void __Method1() {
            Console.WriteLine(y);
        }
    }
}
```

Finally, the following anonymous function captures `this` as well as two local variables with different lifetimes:

```
class Test
{
    int x;

    void F() {
        int y = 123;
        for (int i = 0; i < 10; i++) {
            int z = i * 2;
            D d = () => { Console.WriteLine(x + y + z); };
        }
    }
}
```


www.it-ebooks.info

Here, a compiler-generated class is created for each statement block in which locals are captured, such that the locals in the different blocks can have independent lifetimes. An instance of `__Locals2`, the compiler-generated class for the inner statement block, contains the local variable `z` and a field that references an instance of `__Locals1`. An instance of `__Locals1`, the compiler-generated class for the outer statement block, contains the local variable `y` and a field that references `this` of the enclosing function member. With these data structures, it is possible to reach all captured outer variables through an instance of `__Local2`, and the code of the anonymous function can thus be implemented as an instance method of that class.

BILL WAGNER If `F()` returned an instance of `D`, the lives of all the variables this function captures would be extended.

```
class Test
{
    void F() {
        __Locals1 __locals1 = new __Locals1();
        __locals1.__this = this;
        __locals1.y = 123;
        for (int i = 0; i < 10; i++) {
            __Locals2 __locals2 = new __Locals2();
            __locals2.__locals1 = __locals1;
            __locals2.z = i * 2;
            D d = new D(__locals2.__Method1);
        }
    }

    class __Locals1
    {
        public Test __this;
        public int y;
    }

    class __Locals2
    {
        public __Locals1 __locals1;
        public int z;

        public void __Method1() {
            Console.WriteLine(__locals1.__this.x + __locals1.y + z);
        }
    }
}
```

The same technique applied here to capture local variables can also be used when converting anonymous functions to expression trees: References to the compiler-generated objects can be stored in the expression tree, and access to the local variables can be represented as field accesses on these objects. The advantage of this approach is that it allows the "lifted" local variables to be shared between delegates and expression trees.

www.it-ebooks.info

ERIC LIPPERT A downside of this implementation technique (and a disadvantage shared by many implementations of many languages that have similar closure semantics) is that two different anonymous functions in the same method that are closed over two different local variables cause the lifetimes of both variables to be extended to match the lifetime of the longest-lived delegate.

Suppose you have two local variables, `expensive` and `cheap`, and two delegates, one closed over each variable:

```
D longlived = ()=>cheap;
D shortlived = ()=>expensive;
```

The lifetime of `expensive` is at least as long as the lifetime of the object referred to by `longlived`, even though `expensive` is not actually used by `longlived`. A more sophisticated implementation would partition closed-over variables into separate compiler-generated classes and avoid this problem.

6.6 Method Group Conversions

An implicit conversion (§6.1) exists from a method group (§7.1) to a compatible delegate type. Given a delegate type `D` and an expression `E` that is classified as a method group, an implicit conversion exists from `E` to `D` if `E` contains at least one method that is applicable in its normal form (§7.5.3.1) to an argument list constructed by use of the parameter types and modifiers of `D`, as described in the following.

The compile-time application of a conversion from a method group `E` to a delegate type `D` is described in the following. Note that the existence of an implicit conversion from `E` to `D` does not guarantee that the compile-time application of the conversion will succeed without error.

VLADIMIR RESHETNIKOV In particular, it is possible that overload resolution will pick the best function member containing a delegate type in its signature, while an implicit conversion from a method group, provided as the corresponding argument, to that delegate type will result in a compile-time error. This situation may occur, for instance, because overload resolution cannot find the single best method in that group. Even if the best method is found, constraints on its type parameters might be violated, or the method might not be compatible with the delegate type, or it might be an instance method referenced in a static context.

www.it-ebooks.info

- A single method `M` is selected corresponding to a method invocation (§7.6.5.1) of the form `E(A)`, with the following modifications:
  - The argument list `A` is a list of expressions, each classified as a variable and with the type and modifier (`ref` or `out`) of the corresponding parameter in the *formal-parameter-list* of `D`.
  - The candidate methods considered are only those methods that are applicable in their normal form (§7.5.3.1), not those applicable only in their expanded form.
- If the algorithm of §7.6.5.1 produces an error, then a compile-time error occurs. Otherwise, the algorithm produces a single best method `M` having the same number of parameters as `D` and the conversion is considered to exist.
- The selected method `M` must be compatible (§15.2) with the delegate type `D`; otherwise, a compile-time error occurs.
- If the selected method `M` is an instance method, the instance expression associated with `E` determines the target object of the delegate.
- If the selected method `M` is an extension method, which is denoted by means of a member access on an instance expression, that instance expression determines the target object of the delegate.
- The result of the conversion is a value of type `D`—namely, a newly created delegate that refers to the selected method and target object.

Note that this process can lead to the creation of a delegate to an extension method, if the algorithm of §7.6.5.1 fails to find an instance method but succeeds in processing the invocation of `E(A)` as an extension method invocation (§7.6.5.2). A delegate thus created captures the extension method as well as its first argument.

VLADIMIR RESHETNIKOV If the first parameter of an extension method converted to a delegate type is of a value type or type parameter not known to be a reference type (§10.1.5), a compile-time error occurs.

If an instance method declared in `System.Nullable<T>` is converted to a delegate type, a compile-time error occurs as well.

The following example demonstrates method group conversions:

```
delegate string D1(object o);

delegate object D2(string s);

delegate object D3();

delegate string D4(object o, params object[] a);
```


www.it-ebooks.info

```
delegate string D5(int i);

class Test
{
    static string F(object o) {...}

    static void G() {
        D1 d1 = F;      // Okay
        D2 d2 = F;      // Okay
        D3 d3 = F;      // Error: not applicable
        D4 d4 = F;      // Error: not applicable in normal form
        D5 d5 = F;      // Error: applicable but not compatible

    }
}
```


The assignment to `d1` implicitly converts the method group `F` to a value of type `D1`.

The assignment to `d2` shows how it is possible to create a delegate to a method that has less derived (contravariant) parameter types and a more derived (covariant) return type.

The assignment to `d3` shows how no conversion exists if the method is not applicable.

The assignment to `d4` shows how the method must be applicable in its normal form.

The assignment to `d5` shows how parameter and return types of the delegate and method are allowed to differ only for reference types.

As with all other implicit and explicit conversions, the cast operator can be used to explicitly perform a method group conversion. Thus the example

```
object obj = new EventHandler(myDialog.OkClick);
```

could instead be written

```
object obj = (EventHandler)myDialog.OkClick;
```

Method groups may influence overload resolution, and participate in type inference. See §7.5 for further details.

The runtime evaluation of a method group conversion proceeds as follows:

- If the method selected at compile time is an instance method, or if it is an extension method that is accessed as an instance method, the target object of the delegate is determined from the instance expression associated with `E`:
  - The instance expression is evaluated. If this evaluation causes an exception, no further steps are executed.
  - If the instance expression is of a *reference-type*, the value computed by the instance expression becomes the target object. If the selected method is an instance method

www.it-ebooks.info

and the target object is `null`, a `System.NullReferenceException` is thrown and no further steps are executed.

- If the instance expression is of a *value-type*, a boxing operation (§4.3.1) is performed to convert the value to an object, and this object becomes the target object.

- Otherwise, the selected method is part of a static method call, and the target object of the delegate is `null`.
- A new instance of the delegate type `D` is allocated. If there is not enough memory available to allocate the new instance, a `System.OutOfMemoryException` is thrown and no further steps are executed.

> **JON SKEET** The fact that the specification explicitly states that a new instance is created at this point prevents a potential optimization. The Microsoft C# compiler is able to cache delegates created via anonymous functions if they don't capture any variables (including `this`). The same sort of caching would be feasible for delegates created via any method group conversion that selects a static method—but the specification prohibits it.

- The new delegate instance is initialized with a reference to the method that was determined at compile time and a reference to the target object computed above.

www.it-ebooks.info

This page intentionally left blank

www.it-ebooks.info

7. Expressions

An expression is a sequence of operators and operands. This chapter defines the syntax, order of evaluation of operands and operators, and meaning of expressions.

7.1 Expression Classifications

An expression is classified as one of the following:

- A value. Every value has an associated type.
- A variable. Every variable has an associated type—namely, the declared type of the variable.

> **VLADIMIR RESHETNIKOV** Types of local variables and lambda expression parameters can be inferred by the compiler in many cases, and do not always need to be explicitly specified in declarations.

- A namespace. An expression with this classification can only appear as the left-hand side of a *member-access* (§7.6.4). In any other context, an expression classified as a namespace causes a compile-time error.
- A type. An expression with this classification can only appear as the left-hand side of a *member-access* (§7.6.4), or as an operand for the `as` operator (§7.10.11), the `is` operator (§7.10.10), or the `typeof` operator (§7.6.11). In any other context, an expression classified as a type causes a compile-time error.

> **VLADIMIR RESHETNIKOV** If an expression is classified as a type and appears as the left-hand side of a *member-access*, it never denotes an array or pointer type. If the expression denotes a type parameter, it always leads to a compile-time error later.

www.it-ebooks.info

- A method group, which is a set of overloaded methods resulting from a member lookup (§7.4). A method group may have an associated instance expression and an associated type argument list. When an instance method is invoked, the result of evaluating the instance expression becomes the instance represented by `this` (§7.6.7). A method group is permitted in an *invocation-expression* (§7.6.5), in a *delegate-creation-expression* (§7.6.10.5), and as the left-hand side of an `is` operator, and can be implicitly converted to a compatible delegate type (§6.6). In any other context, an expression classified as a method group causes a compile-time error.

> **ERIC LIPPERT** That a method group is legal on the left-hand side of an `is` operator is a bit of a misleading feature. The result of the `is` evaluation will always be `false`, even if the method group is convertible to the type on the right-hand side.

> **VLADIMIR RESHETNIKOV** A method group on the left-hand side of an `is` operator cannot be used in expression trees.

- A null literal. An expression with this classification can be implicitly converted to a reference type or nullable type.
- An anonymous function. An expression with this classification can be implicitly converted to a compatible delegate type or expression tree type.

> **ERIC LIPPERT** Method groups, anonymous functions, and the null literal are all expressions that have no type. These unusual expressions can be used only when the type can be figured out from the context.

- A property access. Every property access has an associated type—namely, the type of the property. Furthermore, a property access may have an associated instance expression. When an accessor (the `get` or `set` block) of an instance property access is invoked, the result of evaluating the instance expression becomes the instance represented by `this` (§7.6.7).
- An event access. Every event access has an associated type—namely, the type of the event. Furthermore, an event access may have an associated instance expression. An event access may appear as the left-hand operand of the `+=` and `-=` operators (§7.17.3). In any other context, an expression classified as an event access causes a compile-time error.

www.it-ebooks.info

- An indexer access. Every indexer access has an associated type—namely, the element type of the indexer. Furthermore, an indexer access has an associated instance expression and an associated argument list. When an accessor (the `get` or `set` block) of an indexer access is invoked, the result of evaluating the instance expression becomes the instance represented by `this` (§7.6.7), and the result of evaluating the argument list becomes the parameter list of the invocation.
- Nothing. This occurs when the expression is an invocation of a method with a return type of `void`. An expression classified as nothing is only valid in the context of a *statement-expression* (§8.6).

The final result of an expression is never a namespace, type, method group, or event access. Rather, as noted above, these categories of expressions are intermediate constructs that are permitted only in certain contexts.

A property access or indexer access is always reclassified as a value by performing an invocation of the *get-accessor* or the *set-accessor*. The particular accessor is determined by the context of the property or indexer access: If the access is the target of an assignment, the *set-accessor* is invoked to assign a new value (§7.17.1). Otherwise, the *get-accessor* is invoked to obtain the current value (§7.1.1).

7.1.1 Values of Expressions

Most of the constructs that involve an expression ultimately require the expression to denote a ***value***. In such cases, if the actual expression denotes a namespace, a type, a method group, or nothing, a compile-time error occurs. However, if the expression denotes a property access, an indexer access, or a variable, the value of the property, indexer, or variable is implicitly substituted:

- The value of a variable is simply the value currently stored in the storage location identified by the variable. A variable must be considered definitely assigned (§5.3) before its value can be obtained; otherwise, a compile-time error occurs.
- The value of a property access expression is obtained by invoking the *get-accessor* of the property. If the property has no *get-accessor*, a compile-time error occurs. Otherwise, a function member invocation (§7.5.4) is performed, and the result of the invocation becomes the value of the property access expression.
- The value of an indexer access expression is obtained by invoking the *get-accessor* of the indexer. If the indexer has no *get-accessor*, a compile-time error occurs. Otherwise, a function member invocation (§7.5.4) is performed with the argument list associated with the indexer access expression, and the result of the invocation becomes the value of the indexer access expression.

www.it-ebooks.info

> **VLADIMIR RESHETNIKOV** If a property or indexer has a *get-accessor*, but the accessor has an accessibility modifier and is not accessible in the current context, a compile-time error occurs.

7.2 Static and Dynamic Binding

The process of determining the meaning of an operation based on the type or value of constituent expressions (arguments, operands, receivers) is often referred to as ***binding***. For instance, the meaning of a method call is determined based on the type of the receiver and arguments. The meaning of an operator is determined based on the type of its operands.

In C#, the meaning of an operation is usually determined at compile time, based on the compile-time type of its constituent expressions. Likewise, if an expression contains an error, the error is detected and reported by the compiler. This approach is known as ***static binding***.

However, if an expression is a *dynamic expression* (i.e., has the type `dynamic`), this indicates that any binding that it participates in should be based on its runtime type (i.e., the *actual* type of the object it denotes at runtime) rather than the type it has at compile time. The binding of such an operation is therefore deferred until the time where the operation is to be executed during the running of the program. This is referred to as ***dynamic binding***.

When an operation is dynamically bound, little or no checking is performed by the compiler. Instead, if the runtime binding fails, errors are reported as exceptions at runtime.

The following operations in C# are subject to binding:

- Member access: `e.M`
- Method invocation: `e.M(`e_1`,...,`e_n`)`
- Delegate invocation: `e(`e_1`,...,`e_n`)`
- Element access: `e[`e_1`,...,`e_n`]`
- Object creation: `new C(`e_1`,...,`e_n`)`
- Overloaded unary operators: `+`, `-`, `!`, `~`, `++`, `--`, `true`, `false`
- Overloaded binary operators: `+`, `-`, `*`, `/`, `%`, `&`, `&&`, `|`, `||`, `??`, `^`, `<<`, `>>`, `==`, `!=`, `>`, `<`, `>=`, `<=`
- Assignment operators: `=`, `+=`, `-=`, `*=`, `/=`, `%=`, `&=`, `|=`, `^=`, `<<=`, `>>=`
- Implicit and explicit conversions

www.it-ebooks.info

When no dynamic expressions are involved, C# defaults to static binding, which means that the compile-time types of constituent expressions are used in the selection process. However, when one of the constituent expressions in the operations listed above is a dynamic expression, the operation is instead dynamically bound.

7.2.1 Binding Time

Static binding takes place at compile time, whereas dynamic binding takes place at runtime. In the following sections, the term ***binding time*** refers to either compile time or runtime, depending on when the binding takes place.

The following example illustrates the notions of static and dynamic binding and of binding time:

```
object o = 5;
dynamic d = 5;

Console.WriteLine(5); // Static binding to Console.WriteLine(int)
Console.WriteLine(o); // Static binding to Console.WriteLine(object)
Console.WriteLine(d); // Dynamic binding to Console.WriteLine(int)
```

The first two calls are statically bound: The overload of `Console.WriteLine` is picked based on the compile-time type of their argument. Thus the binding time is *compile time*.

The third call is dynamically bound: The overload of `Console.WriteLine` is picked based on the runtime type of its argument. This happens because the argument is a dynamic expression—its compile-time type is `dynamic`. Thus the binding time for the third call is *runtime*.

> **BILL WAGNER** Many people confuse dynamic binding with type inference. Type inference is statically bound. The compiler determines the type at compile time. For example:
>
> ```
> var i = 5;              // i is an int (Compiler performs type inference)
> Console.WriteLine(i);   // Static binding to Console.WriteLine(int)
> ```
>
> The compiler infers that `i` is an integer. All binding on the variable `i` uses static binding.

7.2.2 Dynamic Binding

The purpose of dynamic binding is to allow C# programs to interact with ***dynamic objects***—that is, objects that do not follow the normal rules of the C# type system. Dynamic objects may be objects from other programming languages with different type systems, or they may be objects that are programmatically set up to implement their own binding semantics for different operations.

www.it-ebooks.info

ERIC LIPPERT The primary motivating examples of such objects are (1) objects from dynamic languages such as IronPython, IronRuby, JScript, and so on; (2) objects from "expando" object models that emphasize addition of new properties at runtime, such as the Internet Explorer Document Object Model and other markup-based object models; and (3) legacy COM objects, such as the Microsoft Office object model. The intention here is to make it easier for professional C# developers to interoperate with these systems; our intention is *not* at all to make C# into a dynamic language like JScript.

The mechanism by which a dynamic object implements its own semantics is implementation defined. A given interface—again, implementation defined—is implemented by dynamic objects to signal to the C# runtime that they have special semantics. Thus, whenever operations on a dynamic object are dynamically bound, their own binding semantics, rather than those of C# as specified in this document, take over.

ERIC LIPPERT The mechanism actually chosen for the Microsoft implementation of this feature is the same mechanism used by IronPython and the other DLR languages to make their dynamic analysis and dispatch efficient. A dynamic operation compiles into a call to methods of DLR objects, which then use a special runtime version of the C# expression binder to build expression trees representing the dynamic operation. These expression trees are then compiled, cached, and reused the next time the call site is executed.

While the purpose of dynamic binding is to allow interoperation with dynamic objects, C# allows dynamic binding on all objects, whether they are dynamic or not. This allows for a smoother integration of dynamic objects, as the results of operations on them may not themselves be dynamic objects, but are still of a type unknown to the programmer at compile time. Also, dynamic binding can help eliminate error-prone reflection-based code even when no objects involved are dynamic objects.

ERIC LIPPERT C# already supported a form of dynamic method dispatch: Virtual methods are technically a form of dynamic dispatch because the runtime type of the receiver is used to determine precisely which method to call. Although we do not intend to make C# a dynamic language (as I noted earlier), there is at least one programming problem where dynamic dispatch comes in handy: the "multiple virtual dispatch" problem. This problem arises when you want to choose which method to call based on the runtime types of many of the arguments, not just the receiver. (It is not difficult to implement "double virtual" dispatch with the visitor pattern, but it becomes awkward to implement runtime dispatching on methods with many argument types.)

www.it-ebooks.info

The following sections describe for each construct in the language exactly when dynamic binding is applied, which kind of compile-time checking (if any) is applied, and what the compile-time result and expression classification are.

7.2.3 Types of Constituent Expressions

When an operation is statically bound, the type of a constituent expression (e.g., a receiver, an argument, an index or an operand) is always considered to be the compile-time type of that expression.

When an operation is dynamically bound, the type of a constituent expression is determined in different ways depending on the compile-time type of the constituent expression:

- A constituent expression of compile-time type `dynamic` is considered to have the type of the actual value that the expression evaluates to at runtime.

> **BILL WAGNER** The rule that expressions of compile-time type `dynamic` use the actual runtime type leads to surprising behavior, such as the following compiler error:
>
> ```
> public class Base
> {
>     public Base(dynamic parameter)
>     {
>     }
> }
>
> public class Derived : Base
> {
>     public Derived(dynamic parameter) : base(parameter)
>     {
>     }
> }
> ```
>
> Dynamic dispatch is not allowed during construction, so you must force static dispatch to the base constructor call.

- A constituent expression whose compile-time type is a type parameter is considered to have the type which the type parameter is bound to at runtime.
- Otherwise, the constituent expression is considered to have its compile-time type.

www.it-ebooks.info

7.3 Operators

Expressions are constructed from ***operands*** and ***operators***. The operators of an expression indicate which operations to apply to the operands. Examples of operators include `+`, `-`, `*`, `/`, and `new`. Examples of operands include literals, fields, local variables, and expressions.

There are three kinds of operators:

- Unary operators. The unary operators take one operand and use either prefix notation (such as `-x`) or postfix notation (such as `x++`).
- Binary operators. The binary operators take two operands and all use infix notation (such as `x + y`).
- Ternary operator. Only one ternary operator, `?:`, exists; it takes three operands and uses infix notation (`c? x: y`).

The order of evaluation of operators in an expression is determined by the ***precedence*** and ***associativity*** of the operators (§7.3.1).

Operands in an expression are evaluated from left to right. For example, in `F(i) + G(i++) * H(i)`, method `F` is called using the old value of `i`, then method `G` is called with the old value of `i`, and, finally, method `H` is called with the new value of `i`. This is separate from and unrelated to operator precedence.

Certain operators can be ***overloaded***. Operator overloading permits user-defined operator implementations to be specified for operations where one or both of the operands are of a user-defined class or struct type (§7.3.2).

7.3.1 Operator Precedence and Associativity

When an expression contains multiple operators, the ***precedence*** of the operators controls the order in which the individual operators are evaluated. For example, the expression `x + y * z` is evaluated as `x + (y * z)` because the `*` operator has higher precedence than the binary `+` operator. The precedence of an operator is established by the definition of its associated grammar production. For example, an *additive-expression* consists of a sequence of *multiplicative-expression*s separated by `+` or `-` operators, thus giving the `+` and `-` operators lower precedence than the `*`, `/`, and `%` operators.

The following table summarizes all operators in order of precedence from highest to lowest:

www.it-ebooks.info

Section	Category	Operators
7.6	Primary	`x.y f(x) a[x] x++ x-- new typeof default checked unchecked delegate`
7.7	Unary	`+ - ! ~ ++x --x (T)x`
7.8	Multiplicative	`* / %`
7.8	Additive	`+ -`
7.9	Shift	`<< >>`
7.10	Relational and type testing	`< > <= >= is as`
7.10	Equality	`== !=`
7.11	Logical AND	`&`
7.11	Logical XOR	`^`
7.11	Logical OR	`|`
7.11	Conditional AND	`&&`
7.12	Conditional OR	`||`
7.12	Null coalescing	`??`
7.14	Conditional	`?:`
7.17, 7.15	Assignment and lambda expression	`= *= /= %= += -= <<= >>= &= ^= |= =>`

When an operand occurs between two operators with the same precedence, the associativity of the operators controls the order in which the operations are performed:

- Except for the assignment operators, all binary operators are ***left-associative***, meaning that operations are performed from left to right. For example, `x + y + z` is evaluated as `(x + y) + z`.
- The assignment operators and the conditional operator (`?:`) are ***right-associative***, meaning that operations are performed from right to left. For example, `x = y = z` is evaluated as `x = (y = z)`.

www.it-ebooks.info

Precedence and associativity can be controlled using parentheses. For example, `x + y * z` first multiplies `y` by `z` and then adds the result to `x`, but `(x + y) * z` first adds `x` and `y` and then multiplies the result by `z`.

> **CHRIS SELLS** I'm not a big fan of programs that rely on operator precedence to execute correctly. When the order of execution is in doubt, wrap your expressions in parentheses. The parentheses will not affect the compiled output (except that you might have fixed a bug), but they make the code much easier to understand for the human readers.

> **JESSE LIBERTY** I see no harm in going even further and *always* wrapping expressions in parentheses. *You* may not be in doubt what is intended, but the poor programmer who has to maintain your code ought not have to look up the precedence to make sense of the code. If the number of parentheses becomes confusing in itself, consider breaking your statement into multiple statements using interim temporary variables.

> **ERIC LIPPERT** The relationship between precedence, associativity, parentheses, and order of execution can be confusing. *Operands* are always evaluated on a strictly left-to-right basis. The way that the results of those evaluations are combined is affected by precedence, associativity, and parentheses. It is emphatically *not* the case that `y` and `z` are evaluated before `x`. Yes, the multiplication is computed before the addition, but the evaluation of `x` occurs *before* the multiplication.

7.3.2 Operator Overloading

All unary and binary operators have predefined implementations that are automatically available in any expression. In addition to the predefined implementations, user-defined implementations can be introduced by including `operator` declarations in classes and structs (§10.10). User-defined operator implementations always take precedence over predefined operator implementations: Only when no applicable user-defined operator implementations exist will the predefined operator implementations be considered, as described in §7.3.3 and §7.3.4.

The ***overloadable unary operators*** are:

```
+   -   !   ~   ++   --   true   false
```


www.it-ebooks.info

Although `true` and `false` are not used explicitly in expressions (and therefore are not included in the precedence table in §7.3.1), they are considered operators because they are invoked in several expression contexts: boolean expressions (§7.20) and expressions involving the conditional (§7.14), and conditional logical operators (§7.12).

The ***overloadable binary operators*** are:

```
+   -   *   /   %   &   |   ^   <<   >>   ==   !=   >   <   >=   <=
```

Only the operators listed above can be overloaded. In particular, it is not possible to overload member access, method invocation, or the `=`, `&&`, `||`, `??`, `?:`, `=>`, `checked`, `unchecked`, `new`, `typeof`, `default`, `as`, and `is` operators.

When a binary operator is overloaded, the corresponding assignment operator, if any, is also implicitly overloaded. For example, an overload of operator `*` is also an overload of operator `*=`. This is described further in §7.17.2. Note that the assignment operator itself (`=`) cannot be overloaded. An assignment always performs a simple bitwise copy of a value into a variable.

Cast operations, such as `(T)x`, are overloaded by providing user-defined conversions (§6.4).

Element access, such as `a[x]`, is not considered an overloadable operator. Instead, user-defined indexing is supported through indexers (§10.9).

In expressions, operators are referenced using operator notation, and in declarations, operators are referenced using functional notation. The following table shows the relationship between operator and functional notations for unary and binary operators. In the first entry, *op* denotes any overloadable unary prefix operator. In the second entry, *op* denotes the unary postfix `++` and `--` operators. In the third entry, *op* denotes any overloadable binary operator.

Operator Notation	Functional Notation
op `x`	`operator` *op*`(x)`
`x` *op*	`operator` *op*`(x)`
`x` *op* `y`	`operator` *op*`(x, y)`

User-defined operator declarations always require at least one of the parameters to be of the class or struct type that contains the operator declaration. Thus it is not possible for a user-defined operator to have the same signature as a predefined operator.

www.it-ebooks.info

User-defined operator declarations cannot modify the syntax, precedence, or associativity of an operator. For example, the `/` operator is always a binary operator, always has the precedence level specified in §7.3.1, and is always left-associative.

While it is possible for a user-defined operator to perform any computation it pleases, implementations that produce results other than those that are intuitively expected are strongly discouraged. For example, an implementation of `operator ==` should compare the two operands for equality and return an appropriate `bool` result.

> **BILL WAGNER** In addition to following this rule, you should limit the use of user-defined operators to those times when the operation will be obvious to the vast majority of the developers who use your code.

> **ERIC LIPPERT** The overloadable operators are all mathematical or logical in nature. If you are building a library of mathematical objects such as matrices, vectors, and so on, by all means create user-defined operators. But please resist the temptation to make "cute" operators, such as "a Customer plus an Order produces an Invoice."

The descriptions of individual operators in §7.6 through §7.12 specify the predefined implementations of the operators and any additional rules that apply to each operator. The descriptions make use of the terms ***unary operator overload resolution***, ***binary operator overload resolution***, and ***numeric promotion***, definitions of which are found in the following sections.

7.3.3 Unary Operator Overload Resolution

An operation of the form *op* `x` or `x` *op*, where *op* is an overloadable unary operator, and `x` is an expression of type `X`, is processed as follows:

- The set of candidate user-defined operators provided by `X` for the operation `operator` *op*`(x)` is determined using the rules of §7.3.5.
- If the set of candidate user-defined operators is not empty, then this becomes the set of candidate operators for the operation. Otherwise, the predefined unary `operator` *op* implementations, including their lifted forms, become the set of candidate operators for the operation. The predefined implementations of a given operator are specified in the description of the operator (§7.6 and §7.7).
- The overload resolution rules of §7.5.3 are applied to the set of candidate operators to select the best operator with respect to the argument list `(x)`, and this operator becomes

www.it-ebooks.info

the result of the overload resolution process. If overload resolution fails to select a single best operator, a binding-time error occurs.

7.3.4 Binary Operator Overload Resolution

An operation of the form `x` *op* `y`, where *op* is an overloadable binary operator, `x` is an expression of type `X`, and `y` is an expression of type `Y`, is processed as follows:

- The set of candidate user-defined operators provided by `X` and `Y` for the operation `operator` *op*`(x, y)` is determined. The set consists of the union of the candidate operators provided by `X` and the candidate operators provided by `Y`, each determined using the rules of §7.3.5. If `X` and `Y` are the same type, or if `X` and `Y` are derived from a common base type, then shared candidate operators occur in the combined set only once.
- If the set of candidate user-defined operators is not empty, then this becomes the set of candidate operators for the operation. Otherwise, the predefined binary `operator` *op* implementations, including their lifted forms, become the set of candidate operators for the operation. The predefined implementations of a given operator are specified in the description of the operator (§7.8 through §7.12).
- The overload resolution rules of §7.5.3 are applied to the set of candidate operators to select the best operator with respect to the argument list `(x, y)`, and this operator becomes the result of the overload resolution process. If overload resolution fails to select a single best operator, a binding-time error occurs.

7.3.5 Candidate User-Defined Operators

Given a type `T` and an operation `operator` *op*`(A)`, where *op* is an overloadable operator and `A` is an argument list, the set of candidate user-defined operators provided by `T` for `operator` *op*`(A)` is determined as follows:

- Determine the type T_0. If `T` is a nullable type, T_0 is its underlying type; otherwise, T_0 is equal to `T`.
- For all `operator` *op* declarations in T_0 and all lifted forms of such operators, if at least one operator is applicable (§7.5.3.1) with respect to the argument list `A`, then the set of candidate operators consists of all such applicable operators in T_0.
- Otherwise, if T_0 is `object`, the set of candidate operators is empty.
- Otherwise, the set of candidate operators provided by T_0 is the set of candidate operators provided by the direct base class of T_0, or the effective base class of T_0 if T_0 is a type parameter.

www.it-ebooks.info

7.3.6 Numeric Promotions

Numeric promotion consists of automatically performing certain implicit conversions of the operands of the predefined unary and binary numeric operators. Numeric promotion is not a distinct mechanism, but rather an effect of applying overload resolution to the predefined operators. Numeric promotion specifically does not affect evaluation of user-defined operators, although user-defined operators can be implemented to exhibit similar effects.

As an example of numeric promotion, consider the predefined implementations of the binary * operator:

```
int operator *(int x, int y);
uint operator *(uint x, uint y);
long operator *(long x, long y);
ulong operator *(ulong x, ulong y);
float operator *(float x, float y);
double operator *(double x, double y);
decimal operator *(decimal x, decimal y);
```

When overload resolution rules (§7.5.3) are applied to this set of operators, the effect is to select the first of the operators for which implicit conversions exist from the operand types. For example, for the operation `b * s`, where `b` is a `byte` and `s` is a `short`, overload resolution selects `operator *(int, int)` as the best operator. Thus the effect is that `b` and `s` are converted to `int`, and the type of the result is `int`. Likewise, for the operation `i * d`, where `i` is an `int` and `d` is a `double`, overload resolution selects `operator *(double, double)` as the best operator.

7.3.6.1 *Unary Numeric Promotions*

Unary numeric promotion occurs for the operands of the predefined `+`, `-`, and `~` unary operators. Unary numeric promotion simply consists of converting operands of type `sbyte`, `byte`, `short`, `ushort`, or `char` to type `int`. Additionally, for the unary `-` operator, unary numeric promotion converts operands of type `uint` to type `long`.

7.3.6.2 *Binary Numeric Promotions*

Binary numeric promotion occurs for the operands of the predefined `+`, `-`, `*`, `/`, `%`, `&`, `|`, `^`, `==`, `!=`, `>`, `<`, `>=`, and `<=` binary operators. Binary numeric promotion implicitly converts both operands to a common type which, in case of the nonrelational operators, also becomes the result type of the operation. Binary numeric promotion consists of applying the following rules, in the order they appear here:

- If either operand is of type `decimal`, the other operand is converted to type `decimal`, or a binding-time error occurs if the other operand is of type `float` or `double`.

www.it-ebooks.info

- Otherwise, if either operand is of type `double`, the other operand is converted to type `double`.
- Otherwise, if either operand is of type `float`, the other operand is converted to type `float`.
- Otherwise, if either operand is of type `ulong`, the other operand is converted to type `ulong`, or a binding-time error occurs if the other operand is of type `sbyte`, `short`, `int`, or `long`.
- Otherwise, if either operand is of type `long`, the other operand is converted to type `long`.
- Otherwise, if either operand is of type `uint` and the other operand is of type `sbyte`, `short`, or `int`, both operands are converted to type `long`.
- Otherwise, if either operand is of type `uint`, the other operand is converted to type `uint`.
- Otherwise, both operands are converted to type `int`.

Note that the first rule disallows any operations that mix the `decimal` type with the `double` and `float` types. The rule follows from the fact that there are no implicit conversions between the `decimal` type and the `double` and `float` types.

Also note that it is not possible for an operand to be of type `ulong` when the other operand is of a signed integral type. The reason is that no integral type exists that can represent the full range of `ulong` as well as the signed integral types.

In both of the above cases, a cast expression can be used to explicitly convert one operand to a type that is compatible with the other operand.

In the example

```
decimal AddPercent(decimal x, double percent)
{
    return x * (1.0 + percent / 100.0);
}
```

a binding-time error occurs because a `decimal` cannot be multiplied by a `double`. The error is resolved by explicitly converting the second operand to `decimal`, as follows:

```
decimal AddPercent(decimal x, double percent)
{
    return x * (decimal)(1.0 + percent / 100.0);
}
```


www.it-ebooks.info

JOSEPH ALBAHARI The predefined operators described in this section always promote the 8- and 16-bit integral types—namely, `short`, `ushort`, `sbyte`, and `byte`. A common trap is assigning the result of a calculation on these types back to an 8- or 16-bit integral:

```
byte a = 1, b = 2;
byte c = a + b;      // Compile-time error
```

In this case, the variables `a` and `b` are promoted to `int`, which requires an explicit cast to `byte` for the code to compile:

```
byte a = 1, b = 2;
byte c = (byte)(a + b);
```

7.3.7 Lifted Operators

Lifted operators permit predefined and user-defined operators that operate on non-nullable value types to also be used with nullable forms of those types. Lifted operators are constructed from predefined and user-defined operators that meet certain requirements, as described in the following list:

- For the unary operators

  ```
  +  ++  -  --  !  ~
  ```

  a lifted form of an operator exists if the operand and result types are both non-nullable value types. The lifted form is constructed by adding a single `?` modifier to the operand and result types. The lifted operator produces a null value if the operand is null. Otherwise, the lifted operator unwraps the operand, applies the underlying operator, and wraps the result.

- For the binary operators

  ```
  +  -  *  /  %  &  |  ^  <<  >>
  ```

  a lifted form of an operator exists if the operand and result types are all non-nullable value types. The lifted form is constructed by adding a single `?` modifier to each operand and result type. The lifted operator produces a null value if one or both operands are null (an exception being the `&` and `|` operators of the `bool?` type, as described in §7.11.3). Otherwise, the lifted operator unwraps the operands, applies the underlying operator, and wraps the result.

- For the equality operators

  ```
  ==  !=
  ```

  a lifted form of an operator exists if the operand types are both non-nullable value types and if the result type is `bool`. The lifted form is constructed by adding a single

www.it-ebooks.info

`?` modifier to each operand type. The lifted operator considers two null values equal, and a null value unequal to any non-null value. If both operands are non-null, the lifted operator unwraps the operands and applies the underlying operator to produce the `bool` result.

- For the relational operators

  ```
  <  >  <=  >=
  ```

  a lifted form of an operator exists if the operand types are both non-nullable value types and if the result type is `bool`. The lifted form is constructed by adding a single `?` modifier to each operand type. The lifted operator produces the value `false` if one or both operands are null. Otherwise, the lifted operator unwraps the operands and applies the underlying operator to produce the `bool` result.

BILL WAGNER The default value of any nullable type is neither greater than nor less than any nullable type containing a value.

7.4 Member Lookup

Member lookup is the process whereby the meaning of a name in the context of a type is determined. A member lookup can occur as part of evaluating a *simple-name* (§7.6.2) or a *member-access* (§7.6.4) in an expression. If the *simple-name* or *member-access* occurs as the *primary-expression* of an *invocation-expression* (§7.6.5.1), the member is said to be *invoked*.

If a member is a method or event, or if it is a constant, field, or property of either a delegate type (§15) or the type `dynamic` (§4.7), then the member is said to be *invocable.*

Member lookup considers not only the name of a member, but also the number of type parameters the member has and whether the member is accessible. For the purposes of member lookup, generic methods and nested generic types have the number of type parameters indicated in their respective declarations and all other members have zero type parameters.

VLADIMIR RESHETNIKOV The result of member lookup never contains operators, indexers, explicitly implemented interface members, static constructors, instance constructors, destructors (finalizers), or compiler-generated members. Although a type parameter contributes to the declaration space of its declaring type, it is not a member of that type and, therefore, cannot be a result of member lookup.

Member lookup returns both static and instance members, regardless of the form of access.

www.it-ebooks.info

A member lookup of a name `N` with `K` type parameters in a type `T` is processed as follows:

- First, a set of accessible members named `N` is determined:
  - If `T` is a type parameter, then the set is the union of the sets of accessible members named `N` in each of the types specified as a primary constraint or secondary constraint (§10.1.5) for `T`, along with the set of accessible members named `N` in `object`.
  - Otherwise, the set consists of all accessible (§3.5) members named `N` in `T`, including inherited members and the accessible members named `N` in `object`. If `T` is a constructed type, the set of members is obtained by substituting type arguments as described in §10.3.2. Members that include an `override` modifier are excluded from the set.
- Next, if `K` is zero, all nested types whose declarations include type parameters are removed. If `K` is not zero, all members with a different number of type parameters are removed. Note that when `K` is zero, methods having type parameters are not removed, since the type inference process (§7.5.2) might be able to infer the type arguments.
- Next, if the member is *invoked*, all non-*invocable* members are removed from the set.

VLADIMIR RESHETNIKOV This rule allows you to invoke an extension method even if a non-invocable instance member with the same name exists in the type `T`. For example, you can invoke extension method `Count()` on a collection even if it has an instance property `Count` of type `int`.

- Next, members that are hidden by other members are removed from the set. For every member `S.M` in the set, where `S` is the type in which the member `M` is declared, the following rules are applied:
  - If `M` is a constant, field, property, event, or enumeration member, then all members declared in a base type of `S` are removed from the set.
  - If `M` is a type declaration, then all nontypes declared in a base type of `S` are removed from the set, and all type declarations with the same number of type parameters as `M` declared in a base type of `S` are removed from the set.
  - If `M` is a method, then all nonmethod members declared in a base type of `S` are removed from the set.

VLADIMIR RESHETNIKOV Any members removed from the set on this step can still cause other members to be removed from the set. Thus the order in which the members are processed is not important.

www.it-ebooks.info

- Next, interface members that are hidden by class members are removed from the set. This step has an effect only if `T` is a type parameter and `T` has both an effective base class other than `object` and a non-empty effective interface set (§10.1.5). For every member `S.M` in the set, where `S` is the type in which the member `M` is declared, the following rules are applied if `S` is a class declaration other than `object`:
  - If `M` is a constant, field, property, event, enumeration member, or type declaration, then all members declared in an interface declaration are removed from the set.
  - If `M` is a method, then all nonmethod members declared in an interface declaration are removed from the set, and all methods with the same signature as `M` declared in an interface declaration are removed from the set.
- Finally, having removed hidden members, the result of the lookup is determined:
  - If the set consists of a single member that is not a method, then this member is the result of the lookup.
  - Otherwise, if the set contains only methods, then this group of methods is the result of the lookup.
  - Otherwise, the lookup is ambiguous, and a binding-time error occurs.

VLADIMIR RESHETNIKOV The Microsoft C# compiler is more lenient in the last case. If the set contains both methods and nonmethods, then a warning CS0467 is issued, all nonmethods are discarded, and the group of remaining methods becomes the result of the lookup. This behavior is useful in some COM interoperability scenarios.

For member lookups in types other than type parameters and interfaces, and member lookups in interfaces that are strictly single-inheritance (each interface in the inheritance chain has exactly zero or one direct base interface), the effect of the lookup rules is simply that derived members hide base members with the same name or signature. Such single-inheritance lookups are never ambiguous. The ambiguities that can possibly arise from member lookups in multiple-inheritance interfaces are described in §13.2.5.

7.4.1 Base Types

For purposes of member lookup, a type `T` is considered to have the following base types:

- If `T` is `object`, then `T` has no base type.
- If `T` is an *enum-type*, the base types of `T` are the class types `System.Enum`, `System.ValueType`, and `object`.

www.it-ebooks.info

- If `T` is a *struct-type*, the base types of `T` are the class types `System.ValueType` and `object`.
- If `T` is a *class-type*, the base types of `T` are the base classes of `T`, including the class type `object`.
- If `T` is an *interface-type*, the base types of `T` are the base interfaces of `T` and the class type `object`.

> **ERIC LIPPERT** This point ensures that it is legal to call `ToString()` and the other members of `System.Object` on a value of interface type. At runtime, any value of the interface type will be either null or something that inherits from `System.Object`, so this is a reasonable choice.

- If `T` is an *array-type*, the base types of `T` are the class types `System.Array` and `object`.
- If `T` is a *delegate-type*, the base types of `T` are the class types `System.Delegate` and `object`.

7.5 Function Members

Function members are members that contain executable statements. Function members are always members of types and cannot be members of namespaces. C# defines the following categories of function members:

- Methods
- Properties
- Events
- Indexers
- User-defined operators
- Instance constructors
- Static constructors
- Destructors

Except for destructors and static constructors (which cannot be invoked explicitly), the statements contained in function members are executed through function member invocations. The actual syntax for writing a function member invocation depends on the particular function member category.

www.it-ebooks.info

The argument list (§7.5.1) of a function member invocation provides actual values or variable references for the parameters of the function member.

Invocations of generic methods may employ type inference to determine the set of type arguments to pass to the method. This process is described in §7.5.2.

Invocations of methods, indexers, operators, and instance constructors employ overload resolution to determine which of a candidate set of function members to invoke. This process is described in §7.5.3.

Once a particular function member has been identified at binding-time, possibly through overload resolution, the actual runtime process of invoking the function member is described in §7.5.4.

The following table summarizes the processing that takes place in constructs involving the six categories of function members that can be explicitly invoked. In the following table, `e`, `x`, `y`, and `value` indicate expressions classified as variables or values, `T` indicates an expression classified as a type, `F` is the simple name of a method, and `P` is the simple name of a property.

Construct	Example	Description
Method invocation	`F(x, y)`	Overload resolution is applied to select the best method `F` in the containing class or struct. The method is invoked with the argument list `(x, y)`. If the method is not `static`, the instance expression is `this`.
	`T.F(x, y)`	Overload resolution is applied to select the best method `F` in the class or struct `T`. A binding-time error occurs if the method is not `static`. The method is invoked with the argument list `(x, y)`.
	`e.F(x, y)`	Overload resolution is applied to select the best method `F` in the class, struct, or interface given by the type of `e`. A binding-time error occurs if the method is `static`. The method is invoked with the instance expression `e` and the argument list `(x, y)`.

Continued

www.it-ebooks.info

Construct	Example	Description
Property access	`P`	The `get` accessor of the property `P` in the containing class or struct is invoked. A compile-time error occurs if `P` is write-only. If `P` is not `static`, the instance expression is `this`.
	`P = value`	The `set` accessor of the property `P` in the containing class or struct is invoked with the argument list `(value)`. A compile-time error occurs if `P` is read-only. If `P` is not `static`, the instance expression is `this`.
	`T.P`	The `get` accessor of the property `P` in the class or struct `T` is invoked. A compile-time error occurs if `P` is not `static` or if `P` is write-only.
	`T.P = value`	The `set` accessor of the property `P` in the class or struct `T` is invoked with the argument list `(value)`. A compile-time error occurs if `P` is not `static` or if `P` is read-only.
	`e.P`	The `get` accessor of the property `P` in the class, struct, or interface given by the type of `e` is invoked with the instance expression `e`. A binding-time error occurs if `P` is `static` or if `P` is write-only.
	`e.P = value`	The `set` accessor of the property `P` in the class, struct, or interface given by the type of `e` is invoked with the instance expression `e` and the argument list `(value)`. A binding-time error occurs if `P` is `static` or if `P` is read-only.

www.it-ebooks.info

Construct	Example	Description
Event access	`E += value`	The `add` accessor of the event `E` in the containing class or struct is invoked. If `E` is not static, the instance expression is `this`.
	`E -= value`	The `remove` accessor of the event `E` in the containing class or struct is invoked. If `E` is not static, the instance expression is `this`.
	`T.E += value`	The `add` accessor of the event `E` in the class or struct `T` is invoked. A binding-time error occurs if `E` is not static.
	`T.E -= value`	The `remove` accessor of the event `E` in the class or struct `T` is invoked. A binding-time error occurs if `E` is not static.
	`e.E += value`	The `add` accessor of the event `E` in the class, struct, or interface given by the type of `e` is invoked with the instance expression `e`. A binding-time error occurs if `E` is static.
	`e.E -= value`	The `remove` accessor of the event `E` in the class, struct, or interface given by the type of `e` is invoked with the instance expression `e`. A binding-time error occurs if `E` is static.
Indexer access	`e[x, y]`	Overload resolution is applied to select the best indexer in the class, struct, or interface given by the type of e. The `get` accessor of the indexer is invoked with the instance expression `e` and the argument list `(x, y)`. A binding-time error occurs if the indexer is write-only.
	`e[x, y] = value`	Overload resolution is applied to select the best indexer in the class, struct, or interface given by the type of `e`. The `set` accessor of the indexer is invoked with the instance expression `e` and the argument list `(x, y, value)`. A binding-time error occurs if the indexer is read-only.

Continued

www.it-ebooks.info

Construct	Example	Description
Operator invocation	`-x`	Overload resolution is applied to select the best unary operator in the class or struct given by the type of `x`. The selected operator is invoked with the argument list `(x)`.
	`x + y`	Overload resolution is applied to select the best binary operator in the classes or structs given by the types of x and y. The selected operator is invoked with the argument list `(x, y)`.
Instance constructor invocation	`new T(x, y)`	Overload resolution is applied to select the best instance constructor in the class or struct `T`. The instance constructor is invoked with the argument list `(x, y)`.

7.5.1 Argument Lists

Every function member and delegate invocation includes an argument list that provides actual values or variable references for the parameters of the function member. The syntax for specifying the argument list of a function member invocation depends on the function member category:

- For instance constructors, methods, indexers, and delegates, the arguments are specified as an *argument-list*, as described below. For indexers, when invoking the `set` accessor, the argument list additionally includes the expression specified as the right operand of the assignment operator.

> **VLADIMIR RESHETNIKOV** This additional argument does not participate in overload resolution for indexers.

- For properties, the argument list is empty when invoking the `get` accessor, and consists of the expression specified as the right operand of the assignment operator when invoking the `set` accessor.
- For events, the argument list consists of the expression specified as the right operand of the `+=` or `-=` operator.
- For user-defined operators, the argument list consists of the single operand of the unary operator or the two operands of the binary operator.

www.it-ebooks.info

The arguments of properties (§10.7), events (§10.8), and user-defined operators (§10.10) are always passed as value parameters (§10.6.1.1). The arguments of indexers (§10.9) are always passed as value parameters (§10.6.1.1) or parameter arrays (§10.6.1.4). Reference and output parameters are not supported for these categories of function members.

The arguments of an instance constructor, method, indexer, or delegate invocation are specified as an *argument-list*:

argument-list:
    argument
    argument-list `,` *argument*

argument:
    argument-name$_{opt}$ *argument-value*

argument-name:
    identifier `:`

argument-value:
    expression
    `ref` *variable-reference*
    `out` *variable-reference*

An *argument-list* consists of one or more *argument*s, separated by commas. Each argument consists of an optional *argument-name* followed by an *argument-value*. An *argument* with an *argument-name* is referred to as a ***named argument***, whereas an *argument* without an *argument-name* is a ***positional argument***. It is an error for a positional argument to appear after a named argument in an *argument-list*.

The *argument-value* can take one of the following forms:

- An *expression*, indicating that the argument is passed as a value parameter (§10.6.1.1).
- The keyword `ref` followed by a *variable-reference* (§5.4), indicating that the argument is passed as a reference parameter (§10.6.1.2). A variable must be definitely assigned (§5.3) before it can be passed as a reference parameter.
- The keyword `out` followed by a *variable-reference* (§5.4), indicating that the argument, is passed as an output parameter (§10.6.1.3). A variable is considered definitely assigned (§5.3) following a function member invocation in which the variable is passed as an output parameter.

7.5.1.1 *Corresponding Parameters*

For each argument in an argument list, there has to be a corresponding parameter in the function member or delegate being invoked.

www.it-ebooks.info

The parameter list used in the following is determined as follows:

- For virtual methods and indexers defined in classes, the parameter list is picked from the most specific declaration or override of the function member, starting with the static type of the receiver, and searching through its base classes.

> **ERIC LIPPERT** These rules address the unfortunate (and, one hopes, unlikely) situation where you have a virtual method—say, `void M(int x, int y)`—and an overriding method—say, `void M(int y, int x)`. Does the method call `M(y:1, x:2)` actually call `M(1, 2)` or `M(2, 1)`? Parameter names are not part of the signature of a method and, therefore, can change when overloaded.

- For interface methods and indexers, the parameter list is picked from the most specific definition of the member, starting with the interface type and searching through the base interfaces. If no unique parameter list is found, a parameter list with inaccessible names and no optional parameters is constructed, so that invocations cannot use named parameters or omit optional arguments.
- For partial methods, the parameter list of the defining partial method declaration is used.
- For all other function members and delegates, there is only a single parameter list, which is the one used.

The position of an argument or parameter is defined as the number of arguments or parameters preceding it in the argument list or parameter list.

The corresponding parameters for function member arguments are established as follows:

- Arguments in the *argument-list* of instance constructors, methods, indexers, and delegates:
  - A positional argument where a fixed parameter occurs at the same position in the parameter list corresponds to that parameter.
  - A positional argument of a function member with a parameter array invoked in its normal form corresponds to the parameter array, which must occur at the same position in the parameter list.
  - A positional argument of a function member with a parameter array invoked in its expanded form, where no fixed parameter occurs at the same position in the parameter list, corresponds to an element in the parameter array.

www.it-ebooks.info

  - A named argument corresponds to the parameter of the same name in the parameter list.
  - For indexers, when invoking the `set` accessor, the expression specified as the right operand of the assignment operator corresponds to the implicit `value` parameter of the `set` accessor declaration.
- For properties, when invoking the `get` accessor, there are no arguments. When invoking the `set` accessor, the expression specified as the right operand of the assignment operator corresponds to the implicit `value` parameter of the `set` accessor declaration.
- For user-defined unary operators (including conversions), the single operand corresponds to the single parameter of the operator declaration.
- For user-defined binary operators, the left operand corresponds to the first parameter, and the right operand corresponds to the second parameter of the operator declaration.

7.5.1.2 *Runtime Evaluation of Argument Lists*

During the runtime processing of a function member invocation (§7.5.4), the expressions or variable references of an argument list are evaluated in order, from left to right, as follows:

- For a value parameter, the argument expression is evaluated and an implicit conversion (§6.1) to the corresponding parameter type is performed. The resulting value becomes the initial value of the value parameter in the function member invocation.
- For a reference or output parameter, the variable reference is evaluated and the resulting storage location becomes the storage location represented by the parameter in the function member invocation. If the variable reference given as a reference or output parameter is an array element of a *reference-type*, a runtime check is performed to ensure that the element type of the array is identical to the type of the parameter. If this check fails, a `System.ArrayTypeMismatchException` is thrown.

Methods, indexers, and instance constructors may declare their rightmost parameter to be a parameter array (§10.6.1.4). Such function members are invoked either in their normal form or in their expanded form depending on which is applicable (§7.5.3.1):

- When a function member with a parameter array is invoked in its normal form, the argument given for the parameter array must be a single expression that is implicitly convertible (§6.1) to the parameter array type. In this case, the parameter array acts precisely like a value parameter.
- When a function member with a parameter array is invoked in its expanded form, the invocation must specify zero or more positional arguments for the parameter array,

www.it-ebooks.info

where each argument is an expression that is implicitly convertible (§6.1) to the element type of the parameter array. In this case, the invocation creates an instance of the parameter array type with a length corresponding to the number of arguments, initializes the elements of the array instance with the given argument values, and uses the newly created array instance as the actual argument.

The expressions of an argument list are always evaluated in the order they are written. Thus the example

```
class Test
{
    static void F(int x, int y = -1, int z = -2)
    {
        System.Console.WriteLine("x = {0}, y = {1}, z = {2}", x, y, z);
    }

    static void Main()
    {
        int i = 0;
        F(i++, i++, i++);
        F(z: i++, x: i++);
    }
}
```

produces the output

```
x = 0, y = 1, z = 2
x = 4, y = -1, z = 3
```

The array covariance rules (§12.5) permit a value of an array type `A[]` to be a reference to an instance of an array type `B[]`, provided an implicit reference conversion exists from `B` to `A`. Because of these rules, when an array element of a *reference-type* is passed as a reference or output parameter, a runtime check is required to ensure that the actual element type of the array is *identical* to that of the parameter. In the example

```
class Test
{
    static void F(ref object x) {...}

    static void Main()
    {
        object[] a = new object[10];
        object[] b = new string[10];
        F(ref a[0]);         // Okay
        F(ref b[1]);         // ArrayTypeMismatchException
    }
}
```

the second invocation of `F` causes a `System.ArrayTypeMismatchException` to be thrown because the actual element type of `b` is `string` and not `object`.

www.it-ebooks.info

When a function member with a parameter array is invoked in its expanded form, the invocation is processed exactly as if an array creation expression with an array initializer (§7.6.10.4) was inserted around the expanded parameters. For example, given the declaration

```
void F(int x, int y, params object[] args);
```

the following invocations of the expanded form of the method

```
F(10, 20);
F(10, 20, 30, 40);
F(10, 20, 1, "hello", 3.0);
```

correspond exactly to

```
F(10, 20, new object[] { });
F(10, 20, new object[] { 30, 40 });
F(10, 20, new object[] { 1, "hello", 3.0 });
```

In particular, note that an empty array is created when there are zero arguments given for the parameter array.

When arguments are omitted from a function member with corresponding optional parameters, the default arguments of the function member declaration are implicitly passed. Because these are always constant, their evaluation will not impact the evaluation order of the remaining arguments.

7.5.2 Type Inference

When a generic method is called without specifying type arguments, a ***type inference*** process attempts to infer type arguments for the call. The presence of type inference allows a more convenient syntax to be used for calling a generic method, and allows the programmer to avoid specifying redundant type information. For example, given the method declaration

```
class Chooser
{
    static Random rand = new Random();

    public static T Choose<T>(T first, T second)
    {
        return (rand.Next(2) == 0) ? first : second;
    }
}
```


www.it-ebooks.info

it is possible to invoke the `Choose` method without explicitly specifying a type argument:

```
int i = Chooser.Choose(5, 213);           // Calls Choose<int>
string s = Chooser.Choose("foo", "bar");  // Calls Choose<string>
```

Through type inference, the type arguments `int` and `string` are determined from the arguments to the method.

JON SKEET On occasion, I have attempted to work through the type inference algorithm for specific cases that have not behaved as I expected. I've nearly always regretted it, coming out of the process more confused than before. It feels somewhat like entering a labyrinth, hoping to find the treasure (appropriate type arguments) in the center, but usually encountering a minotaur on the way. I am in awe of those who can not only navigate through the algorithm, but also implement it in a compiler.

CHRIS SELLS If understanding your code feels like navigating a labyrinth, you're doing something wrong.

Type inference occurs as part of the binding-time processing of a method invocation (§7.6.5.1) and takes place before the overload resolution step of the invocation. When a particular method group is specified in a method invocation, and no type arguments are specified as part of the method invocation, type inference is applied to each generic method in the method group. If type inference succeeds, then the inferred type arguments are used to determine the types of arguments for subsequent overload resolution. If overload resolution chooses a generic method as the one to invoke, then the inferred type arguments are used as the actual type arguments for the invocation. If type inference for a particular method fails, that method does not participate in overload resolution. The failure of type inference, in and of itself, does not cause a binding-time error. However, it often leads to a binding-time error when overload resolution then fails to find any applicable methods.

ERIC LIPPERT The type inference algorithm is not guaranteed to produce an applicable candidate. (Of course, overload resolution will weed out the nonapplicable candidates.) The type inference algorithm is designed to answer one question: Given only the arguments and the formal parameter types, what is the *best possible* type argument that can be inferred for each type parameter? If the best possible inference produces an inapplicable candidate, then we do not backtrack and try to guess what the user "really" meant so that inference can choose a different value.

www.it-ebooks.info

MAREK SAFAR When type inference succeeds but overloads resolution fails, the culprit might be an inferred type parameter that uses constraints. The type inference algorithm ignores type parameter constraints, leaving it to overload resolution to verify the best candidate.

If the supplied number of arguments is different than the number of parameters in the method, then inference immediately fails. Otherwise, assume that the generic method has the following signature:

```
T r M<X1...Xn>(T1 x1 ... Tm xm)
```

With a method call of the form `M(E1 ...Em)`, the task of type inference is to find unique type arguments `S1...Sn` for each of the type parameters `X1...Xn` so that the call `M<S1...Sn>(E1...Em)`becomes valid.

VLADIMIR RESHETNIKOV Although it is not indicated explicitly here, the signature can have `ref/out` parameters.

During the process of inference, each type parameter `Xi` is either *fixed* to a particular type `Si` or *unfixed* with an associated set of *bounds*. Each of the bounds is some type `T`. Initially each type variable `Xi` is unfixed with an empty set of bounds.

Type inference takes place in phases. Each phase will try to infer type arguments for more type variables based on the findings of the previous phase. The first phase makes some initial inferences of bounds, whereas the second phase fixes type variables to specific types and infers further bounds. The second phase may have to be repeated a number of times.

Note: Type inference takes place in more situations than just when a generic method is called. Type inference for conversion of method groups is described in §7.5.2.13 and finding the best common type of a set of expressions is described in §7.5.2.14.

ERIC LIPPERT The language design team occasionally is asked, "Why did you invent your own type inference algorithm instead of using a Hindley–Milner-style algorithm?" The short answer is that (1) Hindley–Milner-style algorithms are difficult to adapt to languages with class-based inheritance and covariant types, and (2) backtracking algorithms can potentially take a very long time to execute.

Our type inference algorithm was designed to handle inheritance and type variance. Because every iteration of the second phase either "fixes" an additional type parameter to its inferred type or fails if it is unable to do so, it is clear that the type inference process terminates in fairly short order.

www.it-ebooks.info

PETER SESTOFT Although probably not intended to, the previous annotation may leave the impression that a type inference algorithm for a Hindley–Milner-style type system (that is, ML let polymorphism) must use backtracking. In particular, the Damas–Milner algorithm (1978) does not use backtracking. That said, use of backtracking would be overkill for C# type inference, because in C# all type parameters must be explicitly specified. In contrast, the Damas–Milner algorithm will invent enough type parameters to find the most general ("principal") type for a given expression. That problem is known to be complete for exponential time.

7.5.2.1 *The First Phase*

For each of the method arguments E_i:

- If E_i is an anonymous function, an *explicit parameter type inference* (§7.5.2.7) is made *from* E_i *to* T_i.
- Otherwise, if E_i has a type `U` and x_i is a value parameter, then a *lower-bound inference* is made *from* `U` *to* T_i.
- Otherwise, if E_i has a type `U` and x_i is a `ref` or `out` parameter, then an *exact inference* is made *from* `U` *to* T_i.
- Otherwise, no inference is made for this argument.

7.5.2.2 *The Second Phase*

The second phase proceeds as follows:

- All *unfixed* type variables X_i that do not *depend on* (§7.5.2.5) any X_j are fixed (§7.5.2.10).
- If no such type variables exist, all *unfixed* type variables X_i are *fixed* for which both of the following hold:
  - There is at least one type variable X_j that *depends on* X_i.
  - X_i has a non-empty set of bounds.
- If no such type variables exist and there are still *unfixed* type variables, type inference fails.
- Otherwise, if no further *unfixed* type variables exist, type inference succeeds.
- Otherwise, for all arguments E_i with corresponding parameter type T_i where the *output types* (§7.5.2.4) contain *unfixed* type variables X_j but the *input types* (§7.5.2.3) do not, an *output type inference* (§7.5.2.6) is made *from* E_i *to* T_i. Then the second phase is repeated.

www.it-ebooks.info

7.5.2.3 *Input Types*

If E is a method group or implicitly typed anonymous function and T is a delegate type or expression tree type, then all the parameter types of T are *input types of* E *with type* T.

7.5.2.4 *Output Types*

If E is a method group or an anonymous function and T is a delegate type or expression tree type, then the return type of T is an *output type of* E *with type* T.

7.5.2.5 *Dependence*

An *unfixed* type variable X_i *depends directly on* an *unfixed* type variable X_j if for some argument E_k with type T_k X_j occurs in an *input type* of E_k with type T_k and X_i occurs in an *output type* of E_k with type T_k.

X_j *depends on* X_i if X_j *depends directly on* X_i or if X_i *depends directly on* X_k and X_k *depends on* X_j. Thus "*depends on*" is the transitive but not reflexive closure of "*depends directly on.*"

> **VLADIMIR RESHETNIKOV** A type parameter could possibly directly or indirectly depend on itself.

7.5.2.6 *Output Type Inferences*

An *output type inference* is made *from* an expression E *to* a type T in the following way:

- If E is an anonymous function with inferred return type U (§7.5.2.12) and T is a delegate type or expression tree type with return type T_b, then a *lower-bound inference* (§7.5.2.9) is made *from* U *to* T_b.
- Otherwise, if E is a method group and T is a delegate type or expression tree type with parameter types $T_1 \ldots T_k$ and return type T_b, and overload resolution of E with the types $T_1 \ldots T_k$ yields a single method with return type U, then a *lower-bound inference* is made *from* U *to* T_b.

> **VLADIMIR RESHETNIKOV** This step applies only if all method type parameters occurring in the delegate parameter types are already fixed. Overload resolution does not try to select the best method based on incomplete type information.

- Otherwise, if E is an expression with type U, then a *lower-bound inference* is made *from* U *to* T.
- Otherwise, no inferences are made.

www.it-ebooks.info

7.5.2.7 *Explicit Parameter Type Inferences*

An *explicit parameter type inference* is made *from* an expression `E` *to* a type `T` in the following way:

- If `E` is an explicitly typed anonymous function with parameter types $U_1 \ldots U_k$ and `T` is a delegate type or expression tree type with parameter types $V_1 \ldots V_k$, then for each U_i an *exact inference* (§7.5.2.8) is made *from* U_i *to* the corresponding V_i.

7.5.2.8 *Exact Inferences*

An *exact inference from* a type `U` *to* a type `V` is made as follows:

- If `V` is one of the *unfixed* X_i, then `U` is added to the set of exact bounds for X_i.
- Otherwise, sets $V_1 \ldots V_k$ and $U_1 \ldots U_k$ are determined by checking if any of the following cases apply:
  - `V` is an array type V_1`[...]` and `U` is an array type U_1`[...]` of the same rank.
  - `V` is the type V_1`?` and `U` is the type U_1`?`.
  - `V` is a constructed type `C<`$V_1 \ldots V_k$`>` and `U` is a constructed type `C<`$U_1 \ldots U_k$`>`.
  - If any of these cases apply, then an *exact inference* is made *from* each U_i *to* the corresponding V_i.
- Otherwise, no inferences are made.

7.5.2.9 *Lower-Bound Inferences*

A *lower-bound inference from* a type `U` *to* a type `V` is made as follows:

- If `V` is one of the *unfixed* X_i, then `U` is added to the set of lower bounds for X_i.
- Otherwise, sets $U_1 \ldots U_k$ and $V_1 \ldots V_k$ are determined by checking if any of the following cases apply:
  - `V` is an array type V_1`[...]`and `U` is an array type U_1`[...]` (or a type parameter whose effective base type is U_1`[...]`) of the same rank.
  - `V` is one of `IEnumerable<`V_1`>`, `ICollection<`V_1`>`, or `IList<`V_1`>` and `U` is a one-dimensional array type U_1`[]` (or a type parameter whose effective base type is U_1`[]`).

VLADIMIR RESHETNIKOV Of course, this bullet (and the corresponding bullet in the next paragraph) mentions interfaces from namespace `System.Collections.Generic`.

www.it-ebooks.info

- V is the type V_1? and U is the type U_1?.
- V is a constructed class, struct, interface, or delegate type C<V_1...V_k> and there is a unique type C<U_1...U_k> such that U (or, if U is a type parameter, its effective base class or any member of its effective interface set) is identical to, inherits from (directly or indirectly), or implements (directly or indirectly) C<U_1...U_k>.

  (The "uniqueness" restriction means that in the case `interface C<T>{} class U: C<X>, C<Y>{}`, then no inference is made when inferring from U to C<T> because U_1 could be X or Y.)

VLADIMIR RESHETNIKOV The same inference rule applies to constructed enum types. Although enum types cannot have type parameters in their declarations, they can still be generic if nested within a generic class or struct type.

If any of these cases apply, then an inference is made *from* each U_i *to* the corresponding V_i as follows:

- If U_i is not known to be a reference type, then an *exact inference* is made.
- Otherwise, if U is an array type, then a *lower-bound inference* is made.
- Otherwise, if V is C<V_1...V_k>, then inference depends on the i-th type parameter of C:
  - If it is covariant, then a *lower-bound inference* is made.
  - If it is contravariant, then an *upper-bound inference* is made.
  - If it is invariant, then an *exact inference* is made.
- Otherwise, no inferences are made.

7.5.2.10 *Upper-Bound Inferences*

An *upper-bound inference from* a type U *to* a type V is made as follows:

- If V is one of the *unfixed* X_i, then U is added to the set of upper bounds for X_i.
- Otherwise, sets V_1...V_k and U_1...U_k are determined by checking if any of the following cases apply:
  - U is an array type U_1[...] and V is an array type V_1[...] of the same rank.
  - U is one of IEnumerable<U_e>, ICollection<U_e>, or IList<U_e> and V is a one-dimensional array type V_e[].
  - U is the type U_1? and V is the type V_1?.

www.it-ebooks.info

- `U` is constructed class, struct, interface, or delegate type `C<U1...Uk>` and `V` is a class, struct, interface, or delegate type that is identical to, inherits from (directly or indirectly), or implements (directly or indirectly) a unique type `C<V1...Vk>`.

  (The "uniqueness" restriction means that if we have `interface C<T>{} class V<Z>: C<X<Z>>, C<Y<Z>>{}`, then no inference is made when inferring from `C<U1>` to `V<Q>`. Inferences are not made from `U1` to either `X<Q>` or `Y<Q>`.)

If any of these cases apply, then an inference is made *from* each `Ui` *to* the corresponding `Vi` as follows:

- If `Ui` is not known to be a reference type, then an *exact inference* is made.
- Otherwise, if `V` is an array type, then an *upper-bound inference* is made.
- Otherwise, if `U` is `C<U1...Uk>`, then inference depends on the `i`-th type parameter of `C`:
  - If it is covariant, then an *upper-bound inference* is made.
  - If it is contravariant, then a *lower-bound inference* is made.
  - If it is invariant, then an *exact inference* is made.
- Otherwise, no inferences are made.

VLADIMIR RESHETNIKOV So, no inferences are ever made for method type parameters that do not appear in the parameter types of the generic method. For example, no inferences are made for parameters that appear only in constraints:

```
using System.Collections.Generic;

class C
{
    static void Foo<T,U>(T x) where T : IEnumerable<U> { }

    static void Main()
    {
        Foo(new List<int>());
         // Error CS0411: The type arguments for method
         // 'C.Foo<T,U>(T)' cannot be inferred from the
         // usage.
    }
}
```

7.5.2.11 *Fixing*

An *unfixed* type variable `Xi` with a set of bounds is *fixed* as follows:

- The set of *candidate types* `Uj` starts out as the set of all types in the set of bounds for `Xi`.

www.it-ebooks.info

ERIC LIPPERT This condition illustrates a subtle design point of C#. When faced with the need to choose a "best type" given a set of types, we always choose one of the types in the set. That is, if asked to choose the best type in `{Cat, Dog, Goldfish}`, we fail to find a best type; we never say, "The common base type `Animal` is the best we can do given those three choices" because `Animal` is not one of the choices in the first place.

- We then examine each bound for X_i in turn: For each bound `U` of X_i, all types U_j that are not identical to `U` are removed from the candidate set. For each lower bound `U` of X_i, all types U_j to which there is *not* an implicit conversion from `U` are removed from the candidate set. For each upper bound `U` of X_i, all types U_j from which there is *not* an implicit conversion to `U` are removed from the candidate set.
- If among the remaining candidate types U_j there is a unique type `V` from which there is an implicit conversion to all the other candidate types, then X_i is fixed to `V`.
- Otherwise, type inference fails.

ERIC LIPPERT In C# 2.0, the generic method type inference algorithm failed if two distinct bounds were computed for the same type parameter. In C# 3.0, we have situations such as the `Join` method, which must infer the type of the "key" upon which the two collections are joined. If the key in one collection is, say, `int`, and the corresponding key in the other collection is `int?`, then we have two distinct types. Because every `int` may be converted to `int?`, however, we could allow the ambiguity and resolve it by picking the more general of the two types. C# 4.0 supports generic variance, which complicates the situation further; now we might have upper, lower, and exact bounds on a type parameter.

7.5.2.12 *Inferred Return Type*

The ***inferred return type*** of an anonymous function `F` is used during type inference and overload resolution. The inferred return type can only be determined for an anonymous function where all parameter types are known, either because they are explicitly given, provided through an anonymous function conversion, or inferred during type inference on an enclosing generic method invocation. The inferred return type is determined as follows:

- If the body of `F` is an *expression*, then the inferred return type of `F` is the type of that expression.
- If the body of `F` is a *block* and the set of expressions in the block's `return` statements has a best common type `T` (§7.5.2.14), then the inferred return type of `F` is `T`.

www.it-ebooks.info

> **VLADIMIR RESHETNIKOV** Even `return` statements in unreachable code are included in the set and participate in calculation of the inferred return type. This has a surprising consequence, in that removing unreachable code can change the program behavior.

- Otherwise, a return type cannot be inferred for `E`.

As an example of type inference involving anonymous functions, consider the `Select` extension method declared in the `System.Linq.Enumerable` class:

```
namespace System.Linq
{
    public static class Enumerable
    {
        public static IEnumerable<TResult> Select<TSource, TResult>(
            this IEnumerable<TSource> source,
            Func<TSource, TResult> selector)
        {
            foreach (TSource element in source)
            yield return selector(element);
        }
    }
}
```

Assuming the `System.Linq` namespace was imported with a `using` clause, and given a class `Customer` with a `Name` property of type `string`, the `Select` method can be used to select the names of a list of customers:

```
List<Customer> customers = GetCustomerList();
IEnumerable<string> names = customers.Select(c => c.Name);
```

The extension method invocation (§7.6.5.2) of `Select` is processed by rewriting the invocation to a static method invocation:

```
IEnumerable<string> names = Enumerable.Select(customers, c => c.Name);
```

Since type arguments were not explicitly specified, type inference is used to infer the type arguments. First, the `customers` argument is related to the `source` parameter, inferring `T` to be `Customer`. Then, using the anonymous function type inference process described above, `c` is given type `Customer`, and the expression `c.Name` is related to the return type of the `selector` parameter, inferring `S` to be `string`. Thus the invocation is equivalent to

```
Sequence.Select<Customer,string>(customers, (Customer c) => c.Name)
```

and the result is of type `IEnumerable<string>`.

www.it-ebooks.info

The following example demonstrates how anonymous function type inference allows type information to "flow" between arguments in a generic method invocation. Given the method

```
static Z F<X,Y,Z>(X value, Func<X,Y> f1, Func<Y,Z> f2) {
   return f2(f1(value));
}
```

type inference for the invocation

```
double seconds = F("1:15:30", s => TimeSpan.Parse(s), t => t.TotalSeconds);
```

proceeds as follows: First, the argument `"1:15:30"` is related to the `value` parameter, inferring `X` to be `string`. Then, the parameter of the first anonymous function, `s`, is given the inferred type `string`, and the expression `TimeSpan.Parse(s)` is related to the return type of `f1`, inferring `Y` to be `System.TimeSpan`. Finally, the parameter of the second anonymous function, `t`, is given the inferred type `System.TimeSpan`, and the expression `t.TotalSeconds` is related to the return type of `f2`, inferring `Z` to be `double`. Thus the result of the invocation is of type `double`.

7.5.2.13 *Type Inference for Conversion of Method Groups*

Similar to calls of generic methods, type inference must also be applied when a method group `M` containing a generic method is converted to a given delegate type `D` (§6.6). Given a method

T_r M<X_1...X_n>(T_1 x_1 ... T_m x_m)

and the method group `M` being assigned to the delegate type `D`, the task of type inference is to find type arguments $S_1...S_n$ so that the expression

M<S_1...S_n>

becomes compatible (§15.1) with `D`.

Unlike the type inference algorithm for generic method calls, in this case there are only argument *types*—no argument *expressions*. In particular, there are no anonymous functions and hence no need for multiple phases of inference.

Instead, all X_i are considered *unfixed*, and a *lower-bound inference* is made *from* each argument type U_j of `D` *to* the corresponding parameter type T_j of `M`. If for any of the X_i no bounds were found, type inference fails. Otherwise, all X_i are *fixed* to corresponding S_i, which are the result of type inference.

www.it-ebooks.info

7.5.2.14 *Finding the Best Common Type of a Set of Expressions*

In some cases, a common type needs to be inferred for a set of expressions. In particular, the element types of implicitly typed arrays and the return types of anonymous functions with *block* bodies are found in this way.

Intuitively, given a set of expressions $E_1 \ldots E_m$, this inference should be equivalent to calling a method

```
T_r M<X>(X x_1 ... X x_m)
```

with E_i as arguments.

More precisely, the inference starts out with an *unfixed* type variable `X`. *Output type inferences* are then made *from* each E_i *to* `X`. Finally, `X` is *fixed* and, if successful, the resulting type `S` is the resulting best common type for the expressions. If no such `S` exists, the expressions have no best common type.

7.5.3 Overload Resolution

Overload resolution is a binding-time mechanism for selecting the best function member to invoke given an argument list and a set of candidate function members. Overload resolution selects the function member to invoke in the following distinct contexts within C#:

- Invocation of a method named in an *invocation-expression* (§7.6.5.1).
- Invocation of an instance constructor named in an *object-creation-expression* (§7.6.10.1).
- Invocation of an indexer accessor through an *element-access* (§7.6.6).
- Invocation of a predefined or user-defined operator referenced in an expression (§7.3.3 and §7.3.4).

> **VLADIMIR RESHETNIKOV** The same rules govern selection of an attribute instance constructor in an attribute specification, invocation of an instance constructor in a *constructor-initializer*, and invocation of an indexer through a *base-access*.

Each of these contexts defines the set of candidate function members and the list of arguments in its own unique way, as described in detail in the sections listed above. For example, the set of candidates for a method invocation does not include methods marked `override` (§7.4), and methods in a base class are not candidates if any method in a derived class is applicable (§7.6.5.1).

www.it-ebooks.info

Once the candidate function members and the argument list have been identified, the selection of the best function member is the same in all cases:

- Given the set of applicable candidate function members, the best function member in that set is located. If the set contains only one function member, then that function member is the best function member. Otherwise, the best function member is the one function member that is better than all other function members with respect to the given argument list, provided that each function member is compared to all other function members using the rules in §7.5.3.2. If there is not exactly one function member that is better than all other function members, then the function member invocation is ambiguous and a binding-time error occurs.

The following sections define the exact meanings of the terms ***applicable function member*** and ***better function member***.

> **BILL WAGNER** This section gets very complicated. As you read it, remember that every additional overload you create may contribute to the possible ambiguity in overload resolution. You should limit yourself to the number of overloads that truly make it easier on your users. That approach will keep matters as simple as possible for users of your class and help them ensure that they get the right method.

7.5.3.1 *Applicable Function Member*

> **VLADIMIR RESHETNIKOV** This section also governs applicability of delegates (§7.6.5.3).

A function member is said to be an ***applicable function member*** with respect to an argument list `A` when all of the following are true:

- Each argument in `A` corresponds to a parameter in the function member declaration as described in §7.5.1.1, and any parameter to which no argument corresponds is an optional parameter.
- For each argument in `A`, the parameter passing mode of the argument (i.e., value, `ref`, or `out`) is identical to the parameter passing mode of the corresponding parameter, and
  - For a value parameter or a parameter array, an implicit conversion (§6.1) exists from the argument to the type of the corresponding parameter, or
  - For a `ref` or `out` parameter, the type of the argument is identical to the type of the corresponding parameter. After all, a `ref` or `out` parameter is an alias for the argument passed.

www.it-ebooks.info

A function member that includes a parameter array, if the function member is applicable by the above rules, is said to be applicable in its ***normal form***. If a function member that includes a parameter array is not applicable in its normal form, the function member may instead be applicable in its ***expanded form***:

- The expanded form is constructed by replacing the parameter array in the function member declaration with zero or more value parameters of the element type of the parameter array such that the number of arguments in the argument list `A` matches the total number of parameters. If `A` has fewer arguments than the number of fixed parameters in the function member declaration, the expanded form of the function member cannot be constructed and is thus not applicable.
- Otherwise, the expanded form is applicable if for each argument in `A` the parameter passing mode of the argument is identical to the parameter passing mode of the corresponding parameter, and
  - For a fixed value parameter or a value parameter created by the expansion, an implicit conversion (§6.1) exists from the type of the argument to the type of the corresponding parameter, or
  - For a `ref` or `out` parameter, the type of the argument is identical to the type of the corresponding parameter.

7.5.3.2 *Better Function Member*

For the purposes of determining the better function member, a stripped-down argument list `A` is constructed containing just the argument expressions themselves in the order they appear in the original argument list.

Parameter lists for each of the candidate function members are constructed in the following way:

- The expanded form is used if the function member was applicable only in the expanded form.
- Optional parameters with no corresponding arguments are removed from the parameter list.
- The parameters are reordered so that they occur at the same position as the corresponding argument in the argument list.

Given an argument list `A` with a set of argument expressions { E_1, E_2, …, E_N } and two applicable function members M_P and M_Q with parameter types { P_1, P_2, …, P_N } and { Q_1, Q_2, …, Q_N }, M_P is defined to be a ***better function member*** than M_Q if

www.it-ebooks.info

- For each argument, the implicit conversion from E_X to Q_X is not better than the implicit conversion from E_X to P_X, and
- For at least one argument, the conversion from E_X to P_X is better than the conversion from E_X to Q_X.

When performing this evaluation, if M_P or M_Q is applicable in its expanded form, then P_X or Q_X refers to a parameter in the expanded form of the parameter list.

In case the parameter type sequences $\{P_1, P_2, \ldots, P_N\}$ and $\{Q_1, Q_2, \ldots, Q_N\}$ are equivalent (i.e., each P_i has an identity conversion to the corresponding Q_i), the following tie-breaking rules are applied, in order, to determine the better function member:

- If M_P is a nongeneric method and M_Q is a generic method, then M_P is better than M_Q.
- Otherwise, if M_P is applicable in its normal form and M_Q has a `params` array and is applicable only in its expanded form, then M_P is better than M_Q.
- Otherwise, if M_P has more declared parameters than M_Q, then M_P is better than M_Q. This can occur if both methods have `params` arrays and are applicable only in their expanded forms.
- Otherwise, if all parameters of M_P have a corresponding argument whereas default arguments need to be substituted for at least one optional parameter in M_Q, then M_P is better than M_Q.
- Otherwise, if M_P has more specific parameter types than M_Q, then M_P is better than M_Q. Let $\{R_1, R_2, \ldots, R_N\}$ and $\{S_1, S_2, \ldots, S_N\}$ represent the uninstantiated and unexpanded parameter types of M_P and M_Q. M_P's parameter types are more specific than M_Q's if, for each parameter, R_X is not less specific than S_X, and, for at least one parameter, R_X is more specific than S_X:
  - A type parameter is less specific than a nontype parameter.
  - Recursively, a constructed type is more specific than another constructed type (with the same number of type arguments) if at least one type argument is more specific and no type argument is less specific than the corresponding type argument in the other.
  - An array type is more specific than another array type (with the same number of dimensions) if the element type of the first is more specific than the element type of the second.
- Otherwise, if one member is a non-lifted operator and the other is a lifted operator, the non-lifted one is better.
- Otherwise, neither function member is better.

www.it-ebooks.info

7.5.3.3 *Better Conversion from Expression*

Given an implicit conversion C_1 that converts from an expression `E` to a type T_1, and an implicit conversion C_2 that converts from an expression `E` to a type T_2, C_1 is a ***better conversion*** than C_2 if at least one of the following holds:

- `E` has a type `S` and an identity conversion exists from `S` to T_1 but not from `S` to T_2.
- `E` is not an anonymous function and T_1 is a better conversion target than T_2 (§7.5.3.5).
- `E` is an anonymous function, T_1 is either a delegate type D_1 or an expression tree type `Expression<`D_1`>`, T_2 is either a delegate type D_2 or an expression tree type `Expression<`D_2`>` and one of the following holds:
  - D_1 is a better conversion target than D_2.
  - D_1 and D_2 have identical parameter lists, and one of the following holds:
    - D_1 has a return type Y_1, and D_2 has a return type Y_2, an inferred return type `X` exists for `E` in the context of that parameter list (§7.5.2.12), and the conversion from `X` to Y_1 is better than the conversion from `X` to Y_2.
    - D_1 has a return type `Y`, and D_2 is void returning.

7.5.3.4 *Better Conversion from Type*

Given a conversion C_1 that converts from a type `S` to a type T_1, and a conversion C_2 that converts from a type `S` to a type T_2, C_1 is a ***better conversion*** than C_2 if at least one of the following holds:

- An identity conversion exists from `S` to T_1 but not from `S` to T_2.
- T_1 is a better conversion target than T_2 (§7.5.3.5).

7.5.3.5 *Better Conversion Target*

Given two different types T_1 and T_2, T_1 is a better conversion target than T_2 if at least one of the following holds:

- An implicit conversion from T_1 to T_2 exists, and no implicit conversion from T_2 to T_1 exists.
- T_1 is a signed integral type and T_2 is an unsigned integral type. Specifically:
  - T_1 is `sbyte` and T_2 is `byte`, `ushort`, `uint`, or `ulong`.
  - T_1 is `short` and T_2 is `ushort`, `uint`, or `ulong`.

www.it-ebooks.info

- T_1 is `int` and T_2 is `uint`, or `ulong`.
- T_1 is `long` and T_2 is `ulong`.

7.5.3.6 *Overloading in Generic Classes*

While signatures as declared must be unique, it is possible that substitution of type arguments might result in identical signatures. In such cases, the tie-breaking rules of overload resolution above will pick the most specific member.

The following examples show overloads that are valid and invalid according to this rule:

```
interface I1<T> {...}

interface I2<T> {...}

class G1<U>
{
    int F1(U u);          // Overload resolution for G<int>.F1
    int F1(int i);        // will pick nongeneric

    void F2(I1<U> a);     // Valid overload
    void F2(I2<U> a);
}

class G2<U,V>
{
    void F3(U u, V v);       // Valid, but overload resolution for
    void F3(V v, U u);       // G2<int,int>.F3 will fail

    void F4(U u, I1<V> v);   // Valid, but overload resolution for
    void F4(I1<V> v, U u);   // G2<I1<int>,int>.F4 will fail

    void F5(U u1, I1<V> v2); // Valid overload
    void F5(V v1, U u2);

    void F6(ref U u);        // Valid overload
    void F6(out V v);
}
```

7.5.4 Compile-Time Checking of Dynamic Overload Resolution

For most dynamically bound operations, the set of possible candidates for resolution is unknown at compile time. In certain cases, however, the candidate set is known at compile time:

- Static method calls with dynamic arguments.
- Instance method calls where the receiver is not a dynamic expression.
- Indexer calls where the receiver is not a dynamic expression.
- Constructor calls with dynamic arguments.

www.it-ebooks.info

In these cases, a limited compile-time check is performed for each candidate to see if any of them could possibly apply at runtime. This check consists of the following steps:

- Partial type inference: Any type argument that does not depend directly or indirectly on an argument of type `dynamic` is inferred using the rules of §7.5.2. The remaining type arguments are *unknown.*
- Partial applicability check: Applicability is checked according to §7.5.3.1, but ignoring parameters whose types are *unknown.*

If no candidate passes this test, a compile-time error occurs.

7.5.5 Function Member Invocation

This section describes the process that takes place at runtime to invoke a particular function member. It is assumed that a binding-time process has already determined the particular member to invoke, possibly by applying overload resolution to a set of candidate function members.

For purposes of describing the invocation process, function members are divided into two categories:

- Static function members. These are instance constructors, static methods, static property accessors, and user-defined operators. Static function members are always nonvirtual.
- Instance function members. These are instance methods, instance property accessors, and indexer accessors. Instance function members are either nonvirtual or virtual, and are always invoked on a particular instance. The instance is computed by an instance expression, and it becomes accessible within the function member as `this` (§7.6.7).

The runtime processing of a function member invocation consists of the following steps, where `M` is the function member and, if `M` is an instance member, `E` is the instance expression:

- If `M` is a static function member:
  - The argument list is evaluated as described in §7.5.1.
  - `M` is invoked.
- If `M` is an instance function member declared in a *value-type*:
  - `E` is evaluated. If this evaluation causes an exception, then no further steps are executed.
  - If `E` is not classified as a variable, then a temporary local variable of `E`'s type is created and the value of `E` is assigned to that variable. `E` is then reclassified as a reference to that temporary local variable. The temporary variable is accessible as `this` within `M`,

www.it-ebooks.info

but not in any other way. Thus, only when `E` is a true variable is it possible for the caller to observe the changes that `M` makes to `this`.

> **ERIC LIPPERT** This point illustrates yet another way in which the combination of mutability and copy-by-value semantics can lead to trouble. For example, a `readonly` field is not classified as a variable after the constructor runs. Therefore, an attempt to call a method that mutates the contents of a `readonly` field of `value` type succeeds but actually mutates a copy! Avoid these problems by avoiding mutable `value` types altogether.

- The argument list is evaluated as described in §7.5.1.
- `M` is invoked. The variable referenced by `E` becomes the variable referenced by `this`.

- If `M` is an instance function member declared in a *reference-type*:
  - `E` is evaluated. If this evaluation causes an exception, then no further steps are executed.
  - The argument list is evaluated as described in §7.5.1.
  - If the type of `E` is a *value-type*, a boxing conversion (§4.3.1) is performed to convert `E` to type `object`, and `E` is considered to be of type `object` in the following steps. In this case, `M` could only be a member of `System.Object`.
  - The value of `E` is checked to see if it is valid. If the value of `E` is `null`, a `System.NullReferenceException` is thrown and no further steps are executed.
  - The function member implementation to invoke is determined:
    - If the binding-time type of `E` is an interface, the function member to invoke is the implementation of `M` provided by the runtime type of the instance referenced by `E`. This function member is determined by applying the interface mapping rules (§13.4.4) to determine the implementation of `M` provided by the runtime type of the instance referenced by `E`.
    - Otherwise, if `M` is a virtual function member, the function member to invoke is the implementation of `M` provided by the runtime type of the instance referenced by `E`. This function member is determined by applying the rules for determining the most derived implementation (§10.6.3) of `M` with respect to the runtime type of the instance referenced by `E`.
    - Otherwise, `M` is a nonvirtual function member, and the function member to invoke is `M` itself.
- The function member implementation determined in the step above is invoked. The object referenced by `E` becomes the object referenced by `this`.

www.it-ebooks.info

7.5.5.1 *Invocations on Boxed Instances*

A function member implemented in a *value-type* can be invoked through a boxed instance of that *value-type* in the following situations:

- When the function member is an `override` of a method inherited from type `object` and is invoked through an instance expression of type `object`.
- When the function member is an implementation of an interface function member and is invoked through an instance expression of an *interface-type*.
- When the function member is invoked through a delegate.

In these situations, the boxed instance is considered to contain a variable of the *value-type*, and this variable becomes the variable referenced by `this` within the function member invocation. In particular, when a function member is invoked on a boxed instance, it is possible for the function member to modify the value contained in the boxed instance.

7.6 Primary Expressions

Primary expressions include the simplest forms of expressions.

primary-expression:
primary-no-array-creation-expression
array-creation-expression

primary-no-array-creation-expression:
literal
simple-name
parenthesized-expression
member-access
invocation-expression
element-access
this-access
base-access
post-increment-expression
post-decrement-expression
object-creation-expression
delegate-creation-expression
anonymous-object-creation-expression
typeof-expression
checked-expression
unchecked-expression
default-value-expression
anonymous-method-expression

www.it-ebooks.info

Primary expressions are divided between *array-creation-expressions* and *primary-no-array-creation-expressions*. Treating *array-creation-expression* in this way, rather than listing it along with the other simple expression forms, enables the grammar to disallow potentially confusing code such as

```
object o = new int[3][1];
```

which would otherwise be interpreted as

```
object o = (new int[3])[1];
```

7.6.1 Literals

A *primary-expression* that consists of a *literal* (§2.4.4) is classified as a value.

7.6.2 Simple Names

A *simple-name* consists of an identifier, optionally followed by a type argument list:

simple-name:
identifier *type-argument-list*$_{opt}$

A *simple-name* is either of the form `I` or of the form `I<`A_1`, ..., `A_K`>`, where `I` is a single *identifier* and `<`A_1`, ..., `A_K`>` is an optional *type-argument-list*. When no *type-argument-list* is specified, consider `K` to be zero. The *simple-name* is evaluated and classified as follows:

- If `K` is zero and the *simple-name* appears within a *block*, and if the *block*'s (or an enclosing *block*'s) local variable declaration space (§3.3) contains a local variable, parameter, or constant with name `I`, then the *simple-name* refers to that local variable, parameter, or constant and is classified as a variable or value.

> **VLADIMIR RESHETNIKOV** This rule also applies if the *simple-name* appears within a *constructor-initializer* and matches a name of the containing constructor's parameter.

- If `K` is zero and the *simple-name* appears within the body of a generic method declaration and if that declaration includes a type parameter with name `I`, then the *simple-name* refers to that type parameter.

> **VLADIMIR RESHETNIKOV** This condition always leads to a compile-time error later.

www.it-ebooks.info

- Otherwise, for each instance type `T` (§10.3.1), starting with the instance type of the immediately enclosing type declaration and continuing with the instance type of each enclosing class or struct declaration (if any):
  - If `K` is zero and the declaration of `T` includes a type parameter with name `I`, then the *simple-name* refers to that type parameter.

> **VLADIMIR RESHETNIKOV** This condition always leads to a compile-time error later.

  - Otherwise, if a member lookup (§7.4) of `I` in `T` with `K` type arguments produces a match:
    - If `T` is the instance type of the immediately enclosing class or struct type and the lookup identifies one or more methods, the result is a method group with an associated instance expression of `this`. If a type argument list was specified, it is used in calling a generic method (§7.6.5.1).
    - Otherwise, if `T` is the instance type of the immediately enclosing class or struct type, if the lookup identifies an instance member, and if the reference occurs within the *block* of an instance constructor, an instance method, or an instance accessor, the result is the same as a member access (§7.6.4) of the form `this.I`. This can only happen when `K` is zero.
    - Otherwise, the result is the same as a member access (§7.6.4) of the form `T.I` or `T.I<`A_1`, ..., `A_K`>`. In this case, it is a binding-time error for the *simple-name* to refer to an instance member.
- Otherwise, for each namespace `N`, starting with the namespace in which the *simple-name* occurs, continuing with each enclosing namespace (if any), and ending with the global namespace, the following steps are evaluated until an entity is located:
  - If `K` is zero and `I` is the name of a namespace in `N`, then:
    - If the location where the *simple-name* occurs is enclosed by a namespace declaration for `N` and the namespace declaration contains an *extern-alias-directive* or *using-alias-directive* that associates the name `I` with a namespace or type, then the *simple-name* is ambiguous and a compile-time error occurs.
    - Otherwise, the *simple-name* refers to the namespace named `I` in `N`.

www.it-ebooks.info

- Otherwise, if `N` contains an accessible type having name `I` and `K` type parameters, then:
  - If `K` is zero and the location where the *simple-name* occurs is enclosed by a namespace declaration for `N` and the namespace declaration contains an *extern-alias-directive* or *using-alias-directive* that associates the name `I` with a namespace or type, then the *simple-name* is ambiguous and a compile-time error occurs.
  - Otherwise, the *namespace-or-type-name* refers to the type constructed with the given type arguments.
- Otherwise, if the location where the *simple-name* occurs is enclosed by a namespace declaration for `N`:
  - If `K` is zero and the namespace declaration contains an *extern-alias-directive* or *using-alias-directive* that associates the name `I` with an imported namespace or type, then the *simple-name* refers to that namespace or type.
  - Otherwise, if the namespaces imported by the *using-namespace-directive*s of the namespace declaration contain exactly one type having name `I` and `K` type parameters, then the *simple-name* refers to that type constructed with the given type arguments.
  - Otherwise, if the namespaces imported by the *using-namespace-directive*s of the namespace declaration contain more than one type having name `I` and `K` type parameters, then the *simple-name* is ambiguous and an error occurs.

Note that this entire step is exactly parallel to the corresponding step in the processing of a *namespace-or-type-name* (§3.8).

- Otherwise, the *simple-name* is undefined and a compile-time error occurs.

7.6.2.1 *Invariant Meaning in Blocks*

For each occurrence of a given identifier as a *simple-name* in an expression or declarator, within the local variable declaration space (§3.3) immediately enclosing that occurrence, every other occurrence of the same identifier as a *simple-name* in an expression or declarator must refer to the same entity. This rule ensures that the meaning of a name is always the same within a given block, `switch` block, `for` statement, `foreach` statement, `using`-statement, or anonymous function.

> **ERIC LIPPERT** One of the more subtle desirable consequences of this rule is that it becomes safer to undertake refactorings that involve moving around local variable declarations. Any such refactoring that would cause a simple name to change its semantics will be caught by the compiler.

www.it-ebooks.info

The example

```
class Test
{
    double x;

    void F(bool b) {
        x = 1.0;
        if (b) {
            int x;
            x = 1;
        }
    }
}
```

results in a compile-time error because `x` refers to different entities within the outer block (the extent of which includes the nested block in the `if` statement). In contrast, the example

```
class Test
{
    double x;

    void F(bool b) {
        if (b) {
            x = 1.0;
        }
        else {
            int x;
            x = 1;
        }
    }
}
```

is permitted because the name `x` is never used in the outer block.

Note that the rule of invariant meaning applies only to simple names. It is perfectly valid for the same identifier to have one meaning as a simple name and another meaning as the right operand of a member access (§7.6.4). For example:

```
struct Point
{
    int x, y;

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


www.it-ebooks.info

The example above illustrates a common pattern of using the names of fields as parameter names in an instance constructor. In the example, the simple names `x` and `y` refer to the parameters, but that does not prevent the member access expressions `this.x` and `this.y` from accessing the fields.

7.6.3 Parenthesized Expressions

A *parenthesized-expression* consists of an *expression* enclosed in parentheses.

parenthesized-expression:
(*expression*)

A *parenthesized-expression* is evaluated by evaluating the *expression* within the parentheses. If the *expression* within the parentheses denotes a namespace or type, a compile-time error occurs. Otherwise, the result of the *parenthesized-expression* is the result of the evaluation of the contained *expression*.

7.6.4 Member Access

A *member-access* consists of a *primary-expression*, a *predefined-type*, or a *qualified-alias-member*, followed by a "`.`" token, followed by an *identifier*, optionally followed by a *type-argument-list*.

member-access:
primary-expression `.` *identifier type-argument-list*$_{opt}$
predefined-type `.` *identifier type-argument-list*$_{opt}$
qualified-alias-member `.` *identifier type-argument-list*$_{opt}$

predefined-type: one of

```
bool     byte     char     decimal    double    float    int      long
object   sbyte    short    string     uint      ulong    ushort
```

The *qualified-alias-member* production is defined in §9.7.

A *member-access* is either of the form `E.I` or of the form `E.I<`A_1`, ..., `A_K`>`, where `E` is a *primary-expression*, `I` is a single *identifier,* and `<`A_1`, ..., `A_K`>` is an optional *type-argument-list*. When no *type-argument-list* is specified, consider `K` to be zero.

A *member-access* with a *primary-expression* of type `dynamic` is dynamically bound (§7.2.2). In this case the compiler classifies the member access as a property access of type `dynamic`. The rules below to determine the meaning of the *member-access* are then applied at runtime, using the runtime type instead of the compile-time type of the *primary-expression*. If this runtime classification leads to a method group, then the member access must be the *primary-expression* of an *invocation-expression*.

www.it-ebooks.info

The *member-access* is evaluated and classified as follows:

- If `K` is zero, `E` is a namespace, and `E` contains a nested namespace with name `I`, then the result is that namespace.
- Otherwise, if `E` is a namespace and `E` contains an accessible type having name `I` and `K` type parameters, then the result is that type constructed with the given type arguments.
- If `E` is a *predefined-type* or a *primary-expression* classified as a type, if `E` is not a type parameter, and if a member lookup (§7.4) of `I` in `E` with `K` type parameters produces a match, then `E.I` is evaluated and classified as follows:
  - If `I` identifies a type, then the result is that type constructed with the given type arguments.
  - If `I` identifies one or more methods, then the result is a method group with no associated instance expression. If a type argument list was specified, it is used in calling a generic method (§7.6.5.1).
  - If `I` identifies a `static` property, then the result is a property access with no associated instance expression.
  - If `I` identifies a `static` field:
    - If the field is `readonly` and the reference occurs outside the static constructor of the class or struct in which the field is declared, then the result is a value—namely, the value of the static field `I` in `E`.
    - Otherwise, the result is a variable—namely, the static field `I` in `E`.
  - If `I` identifies a `static` event:
    - If the reference occurs within the class or struct in which the event is declared, and the event was declared without *event-accessor-declarations* (§10.8), then `E.I` is processed exactly as if `I` were a static field.
    - Otherwise, the result is an event access with no associated instance expression.
  - If `I` identifies a constant, then the result is a value—namely, the value of that constant.
  - If `I` identifies an enumeration member, then the result is a value—namely, the value of that enumeration member.
  - Otherwise, `E.I` is an invalid member reference, and a compile-time error occurs.

www.it-ebooks.info

- If `E` is a property access, indexer access, variable, or value, the type of which is `T`, and a member lookup (§7.4) of `I` in `T` with `K` type arguments produces a match, then `E.I` is evaluated and classified as follows:
  - First, if `E` is a property or indexer access, then the value of the property or indexer access is obtained (§7.1.1) and `E` is reclassified as a value.
  - If `I` identifies one or more methods, then the result is a method group with an associated instance expression of `E`. If a type argument list was specified, it is used in calling a generic method (§7.6.5.1).
  - If `I` identifies an instance property, then the result is a property access with an associated instance expression of `E`.
  - If `T` is a *class-type* and `I` identifies an instance field of that *class-type*:
    - If the value of `E` is `null`, then a `System.NullReferenceException` is thrown.
    - Otherwise, if the field is `readonly` and the reference occurs outside an instance constructor of the class in which the field is declared, then the result is a value—namely, the value of the field `I` in the object referenced by `E`.
    - Otherwise, the result is a variable—namely, the field `I` in the object referenced by `E`.
  - If `T` is a *struct-type* and `I` identifies an instance field of that *struct-type*:
    - If `E` is a value, or if the field is `readonly` and the reference occurs outside an instance constructor of the struct in which the field is declared, then the result is a value—namely, the value of the field `I` in the struct instance given by `E`.
    - Otherwise, the result is a variable—namely, the field `I` in the struct instance given by `E`.
  - If `I` identifies an instance event:
    - If the reference occurs within the class or struct in which the event is declared, and the event was declared without *event-accessor-declarations* (§10.8), and the reference does not occur as the left-hand side of a `+=` or `-=` operator, then `E.I` is processed exactly as if `I` was an instance field.
    - Otherwise, the result is an event access with an associated instance expression of `E`.
- Otherwise, an attempt is made to process `E.I` as an extension method invocation (§7.6.5.2). If this fails, `E.I` is an invalid member reference, and a binding-time error occurs.

www.it-ebooks.info

PETER SESTOFT The two bulleted points stating "if the field is `readonly` . . . then the result is a value" have a slightly surprising effect when the field has struct type, and that struct has a mutable field (**not** a recommended combination—see other annotations on this point). Consider the following example:

```
struct S {
    public int x;
    public void SetX() { x = 2; }
}
class C {
    static S s;
    public static void M() { s.SetX(); }
}
```

As expected, after the call `s.SetX()` the struct's field `s.x` will have the value `2`. Now if we add a `readonly` modifier to the declaration of field `s`, then suddenly the call `s.SetX()` has no effect! That is, `s` in the method call is now a value, not a variable, per the rules given earlier; therefore `SetX()` is executed on a *copy* of field `s`, not on field `s` itself. Somewhat strangely, if instead `s` were a local variable of struct type declared in a `using` statement (§8.13), which also has the effect of making `s` immutable, then `s.SetX()` updates `s.x` as expected.

7.6.4.1 *Identical Simple Names and Type Names*

In a member access of the form `E.I`, if `E` is a single identifier, and if the meaning of `E` as a *simple-name* (§7.6.2) is a constant, field, property, local variable, or parameter with the same type as the meaning of `E` as a *type-name* (§3.8), then both possible meanings of `E` are permitted. The two possible meanings of `E.I` are never ambiguous, since `I` must necessarily be a member of the type `E` in both cases. In other words, the rule simply permits access to the static members and nested types of `E` where a compile-time error would otherwise have occurred. For example:

```
struct Color
{
    public static readonly Color White = new Color(...);
    public static readonly Color Black = new Color(...);

    public Color Complement() {...}
}

class A
{
    public Color Color;          // Field Color of type Color

    void F() {
        Color = Color.Black;     // References Color.Black static member
```


www.it-ebooks.info

```
        Color = Color.Complement(); // Invokes Complement() on Color field
    }

    static void G() {
        Color c = Color.White;      // References Color.White static member
    }
}
```

Within the `A` class, those occurrences of the `Color` identifier that reference the `Color` type are underlined, and those that reference the `Color` field are not underlined.

7.6.4.2 *Grammar Ambiguities*

The productions for *simple-name* (§7.6.2) and *member-access* (§7.6.4) can give rise to ambiguities in the grammar for expressions. For example, the statement

```
F(G<A,B>(7));
```

could be interpreted as a call to `F` with two arguments, `G < A` and `B > (7)`. Alternatively, it could be interpreted as a call to `F` with one argument, which is a call to a generic method `G` with two type arguments and one regular argument.

If a sequence of tokens can be parsed (in context) as a *simple-name* (§7.6.2), *member-access* (§7.6.4), or *pointer-member-access* (§18.5.2) ending with a *type-argument-list* (§4.4.1), the token immediately following the closing `>` token is examined. If it is one of

```
( ) ] } : ; , . ? == != | ^
```

then the *type-argument-list* is retained as part of the *simple-name*, *member-access,* or *pointer-member-access,* and any other possible parse of the sequence of tokens is discarded. Otherwise, the *type-argument-list* is not considered to be part of the *simple-name*, *member-access,* or *pointer-member-access*, even if there is no other possible parsing of the sequence of tokens. Note that these rules are not applied when parsing a *type-argument-list* in a *namespace-or-type-name* (§3.8). The statement

```
F(G<A,B>(7));
```

will, according to this rule, be interpreted as a call to `F` with one argument, which is a call to a generic method `G` with two type arguments and one regular argument. The statements

```
F(G < A, B > 7);
F(G < A, B >> 7);
```

will each be interpreted as a call to `F` with two arguments.

www.it-ebooks.info

The statement

```
x = F < A > +y;
```

will be interpreted as including a less than operator, greater than operator, and unary plus operator, as if the statement had been written `x = (F < A) > (+y)`, instead of as a *simple-name* with a *type-argument-list* followed by a binary plus operator. In the statement

```
x = y is C<T> + z;
```

the tokens `C<T>` are interpreted as a *namespace-or-type-name* with a *type-argument-list*.

7.6.5 Invocation Expressions

An *invocation-expression* is used to invoke a method.

invocation-expression:
primary-expression (*argument-list*$_{opt}$)

An *invocation-expression* is dynamically bound (§7.2.2) if at least one of the following holds:

- The *primary-expression* has compile-time type `dynamic`.
- At least one argument of the optional *argument-list* has compile-time type `dynamic` and the *primary-expression* does not have a delegate type.

> **VLADIMIR RESHETNIKOV** One exception to this rule: `ref/out` arguments of type `dynamic` do not cause dynamic binding.
>
> If an *invocation-expression* is dynamically bound and its *primary-expression* denotes a method group that resulted from a *base-access*, then a compile-time error (CS1971) occurs.

In this case the compiler classifies the *invocation-expression* as a value of type `dynamic`. The rules below to determine the meaning of the *invocation-expression* are then applied at run-time, using the runtime type instead of the compile-time type of those of the *primary-expression* and arguments that have the compile-time type `dynamic`. If the *primary-expression* does not have compile-time type `dynamic`, then the method invocation undergoes a limited compile-time check as described in §7.5.4.

The *primary-expression* of an *invocation-expression* must be a method group or a value of a *delegate-type*. If the *primary-expression* is a method group, the *invocation-expression* is a method invocation (§7.6.5.1). If the *primary-expression* is a value of a *delegate-type*, the *invocation-expression* is a delegate invocation (§7.6.5.3). If the *primary-expression* is neither a method group nor a value of a *delegate-type*, a binding-time error occurs.

www.it-ebooks.info

The optional *argument-list* (§7.5.1) provides values or variable references for the parameters of the method.

The result of evaluating an *invocation-expression* is classified as follows:

- If the *invocation-expression* invokes a method or delegate that returns `void`, the result is nothing. An expression that is classified as nothing is permitted only in the context of a *statement-expression* (§8.6) or as the body of a *lambda-expression* (§7.15). Otherwise, a binding-time error occurs.
- Otherwise, the result is a value of the type returned by the method or delegate.

7.6.5.1 *Method Invocations*

For a method invocation, the *primary-expression* of the *invocation-expression* must be a method group. The method group identifies the one method to invoke or the set of overloaded methods from which to choose a specific method to invoke. In the latter case, determination of the specific method to invoke is based on the context provided by the types of the arguments in the *argument-list*.

The binding-time processing of a method invocation of the form `M(A)`, where `M` is a method group (possibly including a *type-argument-list*) and `A` is an optional *argument-list*, consists of the following steps:

- The set of candidate methods for the method invocation is constructed. For each method `F` associated with the method group `M`:
  - If `F` is nongeneric, `F` is a candidate when:
    - `M` has no type argument list, and
    - `F` is applicable with respect to `A` (§7.5.3.1).
  - If `F` is generic and `M` has no type argument list, `F` is a candidate when:
    - Type inference (§7.5.2) succeeds, inferring a list of type arguments for the call, and
    - Once the inferred type arguments are substituted for the corresponding method type parameters, all constructed types in the parameter list of F satisfy their constraints (§4.4.4), and the parameter list of `F` is applicable with respect to `A` (§7.5.3.1).

VLADIMIR RESHETNIKOV Note that only constraints on constructed types appearing in the parameter list are checked during this step. Constraints on the generic method itself are checked later, during the final validation.

www.it-ebooks.info

- If `F` is generic and `M` includes a type argument list, `F` is a candidate when:
  - `F` has the same number of method type parameters as were supplied in the type argument list, and
  - Once the type arguments are substituted for the corresponding method type parameters, all constructed types in the parameter list of F satisfy their constraints (§4.4.4), and the parameter list of `F` is applicable with respect to `A` (§7.5.3.1).
- The set of candidate methods is reduced to contain only methods from the most derived types: For each method `C.F` in the set, where `C` is the type in which the method `F` is declared, all methods declared in a base type of `C` are removed from the set. Furthermore, if `C` is a class type other than `object`, all methods declared in an interface type are removed from the set. (This latter rule takes effect only when the method group was the result of a member lookup on a type parameter having an effective base class other than `object` and a non-empty effective interface set.)

VLADIMIR RESHETNIKOV This rule implies that an applicable method from the most derived type is selected, even if a method with better parameter types exists in a base type, and even if the selected method from the most derived type will not pass the final validation. Also remember that all overrides were removed before this step, during member lookup (§7.4).

```
class Base
{
    public virtual void Foo(int x) { }
}

class Derived : Base
{
    public override void Foo(int x) { }

    static void Foo(object x) { }

    static void Main()
    {
        var d = new Derived();
        d.Foo(1);
         // Error CS0176: Member 'Derived.Foo(object)'
         // cannot be accessed with an instance reference;
         // qualify it with a type name instead
    }
}
```

- If the resulting set of candidate methods is empty, then further processing along the following steps are abandoned, and instead an attempt is made to process the invocation as an extension method invocation (§7.6.5.2). If this fails, then no applicable methods exist, and a binding-time error occurs.

www.it-ebooks.info

- The best method of the set of candidate methods is identified using the overload resolution rules of §7.5.3. If a single best method cannot be identified, the method invocation is ambiguous, and a binding-time error occurs. When performing overload resolution, the parameters of a generic method are considered after substituting the type arguments (supplied or inferred) for the corresponding method type parameters.
- Final validation of the chosen best method is performed:
  - The method is validated in the context of the method group: If the best method is a static method, the method group must have resulted from a *simple-name* or a *member-access* through a type. If the best method is an instance method, the method group must have resulted from a *simple-name*, a *member-access* through a variable or value, or a *base-access*. If neither of these requirements is true, a binding-time error occurs.

VLADIMIR RESHETNIKOV If the best method is an instance method, and the method group has resulted from a *simple-name*, then the *simple-name* must not appear in a static context (i.e., in static members, nested types, instance fields, or field-like event initializers or *constructor-initializer*s); otherwise, a binding-time error occurs.

  - If the best method is a generic method, the type arguments (supplied or inferred) are checked against the constraints (§4.4.4) declared on the generic method. If any type argument does not satisfy the corresponding constraint(s) on the type parameter, a binding-time error occurs.

VLADIMIR RESHETNIKOV If this check succeeds, it automatically guarantees that all constraints on all constructed types in the return type and the body of the method are also satisfied.

ERIC LIPPERT The rule described above has provoked much debate: Why does the compiler go through all the work of type inference, overload resolution, and elimination of inapplicable candidates, only to then say, "I've chosen a method that doesn't work after all"? Why not simply say that methods that don't satisfy their constraints are not even candidates? The reason is subtle but important: A fundamental design principle of C# is that the language does not second-guess the user. If the best possible choice implied by the arguments and parameter types is invalid, then either the user intended the best choice to be used but made some sort of mistake, or the user intended the compiler to choose something else. The safest thing to do is assume the former, rather than possibly guessing wrong.

www.it-ebooks.info

Once a method has been selected and validated at binding time by the above steps, the actual runtime invocation is processed according to the rules of function member invocation described in §7.5.4.

BILL WAGNER The following paragraph is key: Extension methods cannot replace behavior defined by a type author.

The intuitive effect of the resolution rules described above is as follows: To locate the particular method invoked by a method invocation, start with the type indicated by the method invocation and proceed up the inheritance chain until at least one applicable, accessible, non-override method declaration is found. Then perform type inference and overload resolution on the set of applicable, accessible, non-override methods declared in that type and invoke the method thus selected. If no method is found, try instead to process the invocation as an extension method invocation.

ERIC LIPPERT Under this design, even if a "better" method might be found in a base class, any applicable method in a derived class gets priority. The reasoning for this design is twofold.

First, the implementers of the derived class presumably have better information about the desired semantics of this operation on their class than the implementers of the base class did; the derived class exists because it added some functionality or specialization to the base class.

Second, this design avoids one of the "brittle base class" family of problems. Suppose you have code that relies on a call to a method in a derived class. If overload resolution is choosing that method today, then a change to the base class tomorrow should not cause overload resolution to silently and suddenly start choosing the base class method upon recompilation.

Of course, this scheme also introduces the opposite scenario: If you depend on overload resolution to choose a method of the base class, then someone who adds a method to a more derived class can silently change the choice overload resolution upon recompilation as well. However, this "brittle derived class" problem is rarely an issue in real production code; most of the time, you want the more derived method chosen for the reasons given above.

www.it-ebooks.info

JON SKEET While I follow Eric's reasoning for the most part, this design permits an unfortunate situation to occur. If a derived class introduces a new overload of a base class method *and overrides the base class method*, then clearly that derived class is aware of the base class method—and yet that override is still not considered when selecting the method, until the algorithm reaches the base class. For example, consider the following methods in a derived class, with a base class declaring the obvious virtual method:

```
public void Foo(object x) { ... }
public override void Foo(int y) { ... }
```

A call to `Foo(10)` on an expression of the derived type will pick the first method—contrary to the expectations of every non-Microsoft developer I've ever presented this design to.

7.6.5.2 *Extension Method Invocations*

In a method invocation (§7.5.5.1) of one of the forms

expr `.` *identifier* `(` `)`
expr `.` *identifier* `(` *args* `)`
expr `.` *identifier* `<` *typeargs* `>` `(` `)`
expr `.` *identifier* `<` *typeargs* `>` `(` *args* `)`

if the normal processing of the invocation finds no applicable methods, an attempt is made to process the construct as an extension method invocation. If *expr* or any of the *args* has compile-time type `dynamic`, extension methods will not apply.

VLADIMIR RESHETNIKOV The last rule does not apply to `ref`/`out` arguments of type `dynamic`.

The objective is to find the best *type-name* `C`, so that the corresponding static method invocation can take place:

`C` `.` *identifier* `(` *expr* `)`
`C` `.` *identifier* `(` *expr* `,` *args* `)`
`C` `.` *identifier* `<` *typeargs* `>` `(` *expr* `)`
`C` `.` *identifier* `<` *typeargs* `>` `(` *expr* `,` *args* `)`

www.it-ebooks.info

An extension method $C_i.M_j$ is ***eligible*** if:

- C_i is a nongeneric, non-nested class.
- The name of M_j is *identifier.*
- M_j is accessible and applicable when applied to the arguments as a static method as shown above.
- An implicit identity, reference, or boxing conversion exists from *expr* to the type of the first parameter of M_j.

> **ERIC LIPPERT** This rule ensures that making a method that extends `double` does not also extend `int`. It also ensures that no extension methods are defined on anonymous functions or method groups.

The search for `C` proceeds as follows:

- Starting with the closest enclosing namespace declaration, continuing with each enclosing namespace declaration, and ending with the containing compilation unit, successive attempts are made to find a candidate set of extension methods:
  - If the given namespace or compilation unit directly contains nongeneric type declarations C_i with eligible extension methods M_j, then the set of those extension methods is the candidate set.
  - If namespaces imported by using namespace directives in the given namespace or compilation unit directly contain nongeneric type declarations C_i with eligible extension methods M_j, then the set of those extension methods is the candidate set.
- If no candidate set is found in any enclosing namespace declaration or compilation unit, a compile-time error occurs.
- Otherwise, overload resolution is applied to the candidate set as described in (§7.5.3). If no single best method is found, a compile-time error occurs.
- `C` is the type within which the best method is declared as an extension method.

Using `C` as a target, the method call is then processed as a static method invocation (§7.5.4).

> **BILL WAGNER** This processing is somewhat complicated, and it reinforces the recommendation that you should not create duplicate extension methods in different namespaces.

www.it-ebooks.info

PETER SESTOFT It is tempting to think of an extension method as just a funny sort of nonvirtual instance method that is called only when no suitable ordinary instance method is available. Unfortunately, an extension method `SetX(this S s)` on a struct type `S` is not quite the same as an instance method on `S`, because the call `s.SetX()` will be transformed to the call `SetX(s)` that passes struct `s` by value, hence copying it. Any side effect will happen on the copy, not on the original struct `s`—yet another reason never to have mutable fields in struct types.

JON SKEET One of the issues with extension methods is just how easy it is to accidentally import more than you expect or want. I would prefer this process to be more explicit, such as with a new type of using directive:

```
using static System.Linq.Enumerable;
```

This would allow class libraries to expose extension methods without adding to IntelliSense confusion for users who didn't want to use them. Additionally, it would provide a note to code maintainers warning them that they should expect to see certain extension methods used within this compilation unit.

The preceding rules mean that instance methods take precedence over extension methods, that extension methods available in inner namespace declarations take precedence over extension methods available in outer namespace declarations, and that extension methods declared directly in a namespace take precedence over extension methods imported into that same namespace with a using namespace directive. For example:

```
public static class E
{
    public static void F(this object obj, int i) { }

    public static void F(this object obj, string s) { }
}

class A { }

class B
{
    public void F(int i) { }
}

class C
{
    public void F(object obj) { }
}
```


www.it-ebooks.info

```
class X
{
    static void Test(A a, B b, C c) {
        a.F(1);             // E.F(object, int)
        a.F("hello");       // E.F(object, string)

        b.F(1);             // B.F(int)
        b.F("hello");       // E.F(object, string)

        c.F(1);             // C.F(object)
        c.F("hello");       // C.F(object)
    }
}
```

In the example, `B`'s method takes precedence over the first extension method, and `C`'s method takes precedence over both extension methods.

```
public static class C
{
    public static void F(this int i) {
            Console.WriteLine("C.F({0})", i);
        }
    public static void G(this int i) {
            Console.WriteLine("C.G({0})", i);
        }
    public static void H(this int i) {
    Console.WriteLine("C.H({0})", i);
        }
    }

namespace N1
{
    public static class D
    {
        public static void F(this int i) {
        Console.WriteLine("D.F({0})", i);
    }
        public static void G(this int i) {
        Console.WriteLine("D.G({0})", i);
        }
    }
}

namespace N2
{
    using N1;

    public static class E
    {
        public static void F(this int i) {
        Console.WriteLine("E.F({0})", i);
        }
    }
```


www.it-ebooks.info

```
        class Test
        {
            static void Main(string[] args)
            {
                1.F();
                2.G();
                3.H();
            }
        }
    }
```

The output of this example is

```
E.F(1)
D.G(2)
C.H(3)
```

`D.G` takes precedence over `C.G`, and `E.F` takes precedence over both `D.F` and `C.F`.

JOSEPH ALBAHARI One reason for instance methods having higher precedence than extension methods is to ensure that the introduction of extension methods into a program or framework doesn't break existing code.

ERIC LIPPERT The way C# looks for extension methods gives rise to a variation on the "brittle derived class" problem mentioned earlier. If you rely on invocation processing to choose an extension method, but someone introduces an identically named instance method upon the class, then the extension method will no longer be called upon recompilation.

This behavior is not really a problem—it's usually exactly what you want. The method on the class has far more information about the internal structure of the class than the static extension method does, so it should be given precedence.

The Microsoft C# compiler does not give a warning that an extension method exists that would be an alternative to the instance method chosen. If the compiler gave such a warning, there would be no easy way to "fix" the warning other than by putting a `#pragma` directive around it to turn the warning off.

www.it-ebooks.info

JON SKEET I think in this case a warning would actually be the most appropriate course of action, so as to alert developers to the possibility that a method with the same name but a different meaning would now be called with no changes (beyond recompilation) of client code. In many cases—where the whole codebase relying on an extension method can be recompiled, so binary compatibility is not required—the method could be removed entirely or renamed. In other cases it could be converted into a non-extension method, allowing binary compatibility with old code. It feels wrong to completely ignore this change in the results of member lookup.

7.6.5.3 *Delegate Invocations*

For a delegate invocation, the *primary-expression* of the *invocation-expression* must be a value of a *delegate-type*. Furthermore, considering the *delegate-type* to be a function member with the same parameter list as the *delegate-type*, the *delegate-type* must be applicable (§7.5.3.1) with respect to the *argument-list* of the *invocation-expression*.

The runtime processing of a delegate invocation of the form `D(A)`, where `D` is a *primary-expression* of a *delegate-type* and `A` is an optional *argument-list*, consists of the following steps:

- `D` is evaluated. If this evaluation causes an exception, no further steps are executed.
- The value of `D` is checked to be valid. If the value of `D` is `null`, a `System.NullReferenceException` is thrown and no further steps are executed.
- Otherwise, `D` is a reference to a delegate instance. Function member invocations (§7.5.4) are performed on each of the callable entities in the invocation list of the delegate. For callable entities consisting of an instance and instance method, the instance for the invocation is the instance contained in the callable entity.

7.6.6 Element Access

An *element-access* consists of a *primary-no-array-creation-expression*, followed by a "`[`" token, followed by an *argument-list*, followed by a "`]`" token. The *argument-list* consists of one or more *argument*s, separated by commas.

element-access:
    primary-no-array-creation-expression `[` *argument-list* `]`

The *argument-list* of an *element-access* is not allowed to contain `ref` or `out` arguments.

www.it-ebooks.info

An *element-access* is dynamically bound (§7.2.2) if at least one of the following holds:

- The *primary-no-array-creation-expression* has compile-time type `dynamic`.
- At least one expression of the *argument-list* has compile-time type `dynamic` and the *primary-no-array-creation-expression* does not have an array type.

In this case the compiler classifies the *element-access* as a value of type `dynamic`. The rules below to determine the meaning of the *element-access* are then applied at runtime, using the runtime type instead of the compile-time type of those of the *primary-no-array-creation-expression* and *argument-list* expressions that have the compile-time type `dynamic`. If the *primary-no-array-creation-expression* does not have compile-time type `dynamic`, then the element access undergoes a limited compile time check as described in §7.5.4.

If the *primary-no-array-creation-expression* of an *element-access* is a value of an *array-type*, the *element-access* is an array access (§7.6.6.1). Otherwise, the *primary-no-array-creation-expression* must be a variable or value of a class, struct, or interface type that has one or more indexer members, in which case the *element-access* is an indexer access (§7.6.6.2).

7.6.6.1 *Array Access*

For an array access, the *primary-no-array-creation-expression* of the *element-access* must be a value of an *array-type*. Furthermore, the *argument-list* of an array access is not allowed to contain named arguments. The number of expressions in the *argument-list* must be the same as the rank of the *array-type*, and each expression must be of type `int`, `uint`, `long`, or `ulong`, or must be implicitly convertible to one or more of these types.

The result of evaluating an array access is a variable of the element type of the array—namely, the array element selected by the value(s) of the expression(s) in the *argument-list*.

> **JON SKEET** The fact that the result is a variable is important here: It means you can use array elements as `ref` or `out` arguments in method calls. In the face of array covariance, the *actual* type of the storage location is validated before the method is called.

The runtime processing of an array access of the form `P[A]`, where `P` is a *primary-no-array-creation-expression* of an *array-type* and `A` is an *argument-list*, consists of the following steps:

- `P` is evaluated. If this evaluation causes an exception, no further steps are executed.
- The index expressions of the *argument-list* are evaluated in order, from left to right. Following evaluation of each index expression, an implicit conversion (§6.1) to one of the following types is performed: `int`, `uint`, `long`, `ulong`. The first type in this list for which

www.it-ebooks.info

an implicit conversion exists is chosen. For instance, if the index expression is of type `short`, then an implicit conversion to `int` is performed, since implicit conversions from `short` to `int` and from `short` to `long` are possible. If evaluation of an index expression or the subsequent implicit conversion causes an exception, then no further index expressions are evaluated and no further steps are executed.

- The value of `P` is checked to be valid. If the value of `P` is `null`, a `System.NullReferenceException` is thrown and no further steps are executed.
- The value of each expression in the *argument-list* is checked against the actual bounds of each dimension of the array instance referenced by `P`. If one or more values are out of range, a `System.IndexOutOfRangeException` is thrown and no further steps are executed.
- The location of the array element given by the index expression(s) is computed, and this location becomes the result of the array access.

7.6.6.2 *Indexer Access*

For an indexer access, the *primary-no-array-creation-expression* of the *element-access* must be a variable or value of a class, struct, or interface type, and this type must implement one or more indexers that are applicable with respect to the *argument-list* of the *element-access*.

The binding-time processing of an indexer access of the form `P[A]`, where `P` is a *primary-no-array-creation-expression* of a class, struct, or interface type `T`, and `A` is an *argument-list*, consists of the following steps:

- The set of indexers provided by `T` is constructed. The set consists of all indexers declared in `T` or a base type of `T` that are not `override` declarations and are accessible in the current context (§3.5).
- The set is reduced to those indexers that are applicable and not hidden by other indexers. The following rules are applied to each indexer `S.I` in the set, where `S` is the type in which the indexer `I` is declared:
  - If `I` is not applicable with respect to `A` (§7.5.3.1), then `I` is removed from the set.
  - If `I` is applicable with respect to `A` (§7.5.3.1), then all indexers declared in a base type of `S` are removed from the set.
  - If `I` is applicable with respect to `A` (§7.5.3.1) and `S` is a class type other than `object`, all indexers declared in an interface are removed from the set.
- If the resulting set of candidate indexers is empty, then no applicable indexers exist, and a binding-time error occurs.

www.it-ebooks.info

- The best indexer of the set of candidate indexers is identified using the overload resolution rules of §7.5.3. If a single best indexer cannot be identified, the indexer access is ambiguous, and a binding-time error occurs.
- The index expressions of the *argument-list* are evaluated in order, from left to right. The result of processing the indexer access is an expression classified as an indexer access. The indexer access expression references the indexer determined in the step above, and has an associated instance expression of `P` and an associated argument list of `A`.

Depending on the context in which it is used, an indexer access causes invocation of either the *get-accessor* or the *set-accessor* of the indexer. If the indexer access is the target of an assignment, the *set-accessor* is invoked to assign a new value (§7.17.1). In all other cases, the *get-accessor* is invoked to obtain the current value (§7.1.1).

VLADIMIR RESHETNIKOV If the corresponding accessor is missing or is not accessible, a compile-time error occurs.

7.6.7 this Access

A *this-access* consists of the reserved word `this`.

this-access:
    `this`

A *this-access* is permitted only in the *block* of an instance constructor, an instance method, or an instance accessor. It has one of the following meanings:

- When `this` is used in a *primary-expression* within an instance constructor of a class, it is classified as a value. The type of the value is the instance type (§10.3.1) of the class within which the usage occurs, and the value is a reference to the object being constructed.
- When `this` is used in a *primary-expression* within an instance method or instance accessor of a class, it is classified as a value. The type of the value is the instance type (§10.3.1) of the class within which the usage occurs, and the value is a reference to the object for which the method or accessor was invoked.
- When `this` is used in a *primary-expression* within an instance constructor of a struct, it is classified as a variable. The type of the variable is the instance type (§10.3.1) of the struct within which the usage occurs, and the variable represents the struct being constructed. The `this` variable of an instance constructor of a struct behaves exactly the same as an `out` parameter of the struct type—in particular, this means that the variable must be definitely assigned in every execution path of the instance constructor.

www.it-ebooks.info

- When `this` is used in a *primary-expression* within an instance method or instance accessor of a struct, it is classified as a variable. The type of the variable is the instance type (§10.3.1) of the struct within which the usage occurs.
  - If the method or accessor is not an iterator (§10.14), the `this` variable represents the struct for which the method or accessor was invoked, and behaves exactly the same as a `ref` parameter of the struct type.
  - If the method or accessor is an iterator, the `this` variable represents a *copy* of the struct for which the method or accessor was invoked, and behaves exactly the same as a *value* parameter of the struct type.

> **JON SKEET** The ability to write
>
> ```
> this = new CustomStruct(...);
> ```
>
> in a method within a value type always feels deeply wrong to me. Given that structs should almost always be immutable in the first place, I wonder how many justifiable uses this has found in the global corpus of C# code.

Use of `this` in a *primary-expression* in a context other than the ones listed above is a compile-time error. In particular, it is not possible to refer to `this` in a static method, a static property accessor, or in a *variable-initializer* of a field declaration.

7.6.8 Base Access

A *base-access* consists of the reserved word `base` followed by either a "`.`" token and an identifier or an *argument-list* enclosed in square brackets:

base-access:
    base `.` *identifier*
    base `[` *argument-list* `]`

A *base-access* is used to access base class members that are hidden by similarly named members in the current class or struct. A *base-access* is permitted only in the *block* of an instance constructor, an instance method, or an instance accessor. When `base.I` occurs in a class or struct, `I` must denote a member of the base class of that class or struct. Likewise, when `base[E]` occurs in a class, an applicable indexer must exist in the base class.

At binding time, *base-access* expressions of the form `base.I` and `base[E]` are evaluated exactly as if they were written `((B)this).I` and `((B)this)[E]`, where `B` is the base class of the class or struct in which the construct occurs. Thus `base.I` and `base[E]` correspond to `this.I` and `this[E]`, except `this` is viewed as an instance of the base class.

www.it-ebooks.info

VLADIMIR RESHETNIKOV If a *base-access* of the latter form is dynamically dispatched (i.e., if it has an argument of type `dynamic`), a compile-time error (CS1972) occurs.

When a *base-access* references a virtual function member (a method, property, or indexer), the determination of which function member to invoke at runtime (§7.5.4) is changed. The function member that is invoked is determined by finding the most derived implementation (§10.6.3) of the function member with respect to `B` (instead of with respect to the runtime type of `this`, as would be usual in a non-base access). Thus, within an `override` of a `virtual` function member, a *base-access* can be used to invoke the inherited implementation of the function member. If the function member referenced by a *base-access* is abstract, a binding-time error occurs.

ERIC LIPPERT In the Microsoft C# 2.0 compiler and above, base calls to virtual methods are code generated as nonvirtual calls to the *specific* method known at compile time to be on a base class. If you do an end-run around the compiler by swapping in a new version of a library that has a new virtual override "in the middle" without recompiling the code that makes the base call, that code will continue to call the specific method identified at compile time. In short, *base calls do not use virtual dispatch.*

It is not legal for a lambda expression converted to an expression tree type to contain a base access.

7.6.9 Postfix Increment and Decrement Operators

post-increment-expression:
    primary-expression `++`

post-decrement-expression:
    primary-expression `--`

The operand of a postfix increment or decrement operation must be an expression classified as a variable, a property access, or an indexer access. The result of the operation is a value of the same type as the operand.

If the *primary-expression* has the compile-time type `dynamic`, then the operator is dynamically bound (§7.2.2), the *post-increment-expression* or *post-decrement-expression* has the compile-time type `dynamic`, and the following rules are applied at runtime using the runtime type of the *primary-expression*.

www.it-ebooks.info

If the operand of a postfix increment or decrement operation is a property or indexer access, the property or indexer must have both a `get` and a `set` accessor. If this is not the case, a binding-time error occurs.

Unary operator overload resolution (§7.3.3) is applied to select a specific operator implementation. Predefined `++` and `--` operators exist for the following types: `sbyte`, `byte`, `short`, `ushort`, `int`, `uint`, `long`, `ulong`, `char`, `float`, `double`, `decimal`, and any enum type. The predefined `++` operators return the value produced by adding 1 to the operand, and the predefined `--` operators return the value produced by subtracting 1 from the operand. In a `checked` context, if the result of this addition or subtraction is outside the range of the result type and the result type is an integral type or enum type, a `System.OverflowException` is thrown.

The runtime processing of a postfix increment or decrement operation of the form `x++` or `x--` consists of the following steps:

- If `x` is classified as a variable:
  - `x` is evaluated to produce the variable.
  - The value of `x` is saved.
  - The selected operator is invoked with the saved value of `x` as its argument.
  - The value returned by the operator is stored in the location given by the evaluation of `x`.
  - The saved value of `x` becomes the result of the operation.
- If `x` is classified as a property or indexer access:
  - The instance expression (if `x` is not `static`) and the argument list (if `x` is an indexer access) associated with `x` are evaluated, and the results are used in the subsequent `get` and `set` accessor invocations.
  - The `get` accessor of `x` is invoked and the returned value is saved.
  - The selected operator is invoked with the saved value of `x` as its argument.
  - The `set` accessor of `x` is invoked with the value returned by the operator as its `value` argument.

VLADIMIR RESHETNIKOV If either the *get-accessor* or the *set-accessor* is missing or is not accessible, a compile-time error occurs.

  - The saved value of `x` becomes the result of the operation.

www.it-ebooks.info

The ++ and -- operators also support prefix notation (§7.7.5). Typically, the result of x++ or x-- is the value of x *before* the operation, whereas the result of ++x or --x is the value of x *after* the operation. In either case, x itself has the same value after the operation.

An operator ++ or operator -- implementation can be invoked using either postfix or prefix notation. It is not possible to have separate operator implementations for the two notations.

7.6.10 The new Operator

The new operator is used to create new instances of types.

There are three forms of new expressions:

- Object creation expressions are used to create new instances of class types and value types.
- Array creation expressions are used to create new instances of array types.
- Delegate creation expressions are used to create new instances of delegate types.

The new operator implies creation of an instance of a type, but does not necessarily imply dynamic allocation of memory. In particular, instances of value types require no additional memory beyond the variables in which they reside, and no dynamic allocations occur when new is used to create instances of value types.

7.6.10.1 *Object Creation Expressions*

An *object-creation-expression* is used to create a new instance of a *class-type* or a *value-type*.

object-creation-expression:
    new *type* (*argument-list*$_{opt}$) *object-or-collection-initializer*$_{opt}$
    new *type* *object-or-collection-initializer*

object-or-collection-initializer:
    object-initializer
    collection-initializer

The *type* of an *object-creation-expression* must be a *class-type*, a *value-type*, or a *type-parameter*. The *type* cannot be an abstract *class-type*.

The optional *argument-list* (§7.5.1) is permitted only if the *type* is a *class-type* or a *struct-type*.

An object creation expression can omit the constructor argument list and enclosing parentheses provided it includes an object initializer or collection initializer. Omitting the constructor argument list and enclosing parentheses is equivalent to specifying an empty argument list.

www.it-ebooks.info

Processing of an object creation expression that includes an object initializer or collection initializer consists of first processing the instance constructor and then processing the member or element initializations specified by the object initializer (§7.6.10.2) or collection initializer (§7.6.10.3).

If any of the arguments in the optional *argument-list* has the compile-time type `dynamic`, then the *object-creation-expression* is dynamically bound (§7.2.2) and the following rules are applied at runtime using the runtime type of those arguments of the *argument-list* that have the compile-time type `dynamic`. However, the object creation undergoes a limited compile-time check, as described in §7.5.4.

The binding-time processing of an *object-creation-expression* of the form `new T(A)`, where `T` is a *class-type* or a *value-type* and `A` is an optional *argument-list*, consists of the following steps:

- If `T` is a *value-type* and `A` is not present:
  - The *object-creation-expression* is a default constructor invocation. The result of the *object-creation-expression* is a value of type `T`—namely, the default value for `T` as defined in §4.1.1.
- Otherwise, if `T` is a *type-parameter* and `A` is not present:
  - If no value type constraint or constructor constraint (§10.1.5) has been specified for `T`, a binding-time error occurs.
  - The result of the *object-creation-expression* is a value of the runtime type that the type parameter has been bound to—namely, the result of invoking the default constructor of that type. The runtime type may be a reference type or a value type.
- Otherwise, if `T` is a *class-type* or a *struct-type*:
  - If `T` is an `abstract` *class-type*, a compile-time error occurs.
  - The instance constructor to invoke is determined using the overload resolution rules of §7.5.3. The set of candidate instance constructors consists of all accessible instance constructors declared in `T` that are applicable with respect to `A` (§7.5.3.1). If the set of candidate instance constructors is empty, or if a single best instance constructor cannot be identified, a binding-time error occurs.
  - The result of the *object-creation-expression* is a value of type `T`—namely, the value produced by invoking the instance constructor determined in the step above.
- Otherwise, the *object-creation-expression* is invalid, and a binding-time error occurs.

Even if the *object-creation-expression* is dynamically bound, the compile-time type is still `T`.

The runtime processing of an *object-creation-expression* of the form `new T(A)`, where `T` is *class-type* or a *struct-type* and `A` is an optional *argument-list*, consists of the following steps:

www.it-ebooks.info

- If T is a *class-type*:
  - A new instance of class T is allocated. If there is not enough memory available to allocate the new instance, a System.OutOfMemoryException is thrown and no further steps are executed.
  - All fields of the new instance are initialized to their default values (§5.2).
  - The instance constructor is invoked according to the rules of function member invocation (§7.5.4). A reference to the newly allocated instance is automatically passed to the instance constructor and the instance can be accessed from within that constructor as this.
- If T is a *struct-type*:
  - An instance of type T is created by allocating a temporary local variable. Since an instance constructor of a *struct-type* is required to definitely assign a value to each field of the instance being created, no initialization of the temporary variable is necessary.
  - The instance constructor is invoked according to the rules of function member invocation (§7.5.4). A reference to the newly allocated instance is automatically passed to the instance constructor and the instance can be accessed from within that constructor as this.

7.6.10.2 *Object Initializers*

An ***object initializer*** specifies values for zero or more fields or properties of an object.

object-initializer:
{ *member-initializer-list*$_{opt}$ }
{ *member-initializer-list* , }

member-initializer-list:
member-initializer
member-initializer-list , *member-initializer*

member-initializer:
identifier = *initializer-value*

initializer-value:
expression
object-or-collection-initializer

An object initializer consists of a sequence of member initializers, enclosed by { and } tokens and separated by commas. Each member initializer must name an accessible field or property of the object being initialized, followed by an equals sign and an expression, object initializer, or collection initializer. It is an error for an object initializer to include

www.it-ebooks.info

more than one member initializer for the same field or property. It is not possible for the object initializer to refer to the newly created object it is initializing.

A member initializer that specifies an expression after the equals sign is processed in the same way as an assignment (§7.17.1) to the field or property.

A member initializer that specifies an object initializer after the equals sign is a ***nested object initializer***—that is, an initialization of an embedded object. Instead of assigning a new value to the field or property, the assignments in the nested object initializer are treated as assignments to members of the field or property. Nested object initializers cannot be applied to properties with a value type, or to read-only fields with a value type.

A member initializer that specifies a collection initializer after the equals sign is an initialization of an embedded collection. Instead of assigning a new collection to the field or property, the elements given in the initializer are added to the collection referenced by the field or property. The field or property must be of a collection type that satisfies the requirements specified in §7.6.10.3.

The following class represents a point with two coordinates:

```
public class Point
{
    int x, y;

    public int X { get { return x; } set { x = value; } }
    public int Y { get { return y; } set { y = value; } }
}
```

An instance of `Point` can be created and initialized as follows:

```
Point a = new Point { X = 0, Y = 1 };
```

which has the same effect as

```
Point __a = new Point();
__a.X = 0;
__a.Y = 1;
Point a = __a;
```

where `__a` is an otherwise invisible and inaccessible temporary variable. The following class represents a rectangle created from two points:

```
public class Rectangle
{
    Point p1, p2;

    public Point P1 { get { return p1; } set { p1 = value; } }
    public Point P2 { get { return p2; } set { p2 = value; } }
}
```

An instance of `Rectangle` can be created and initialized as follows:

www.it-ebooks.info

```
Rectangle r = new Rectangle {
    P1 = new Point { X = 0, Y = 1 },
    P2 = new Point { X = 2, Y = 3 }
};
```

which has the same effect as

```
Rectangle __r = new Rectangle();
Point __p1 = new Point();
__p1.X = 0;
__p1.Y = 1;
__r.P1 = __p1;
Point __p2 = new Point();
__p2.X = 2;
__p2.Y = 3;
__r.P2 = __p2;
Rectangle r = __r;
```

where `__r`, `__p1`, and `__p2` are temporary variables that are otherwise invisible and inaccessible.

JOSEPH ALBAHARI The use of a hidden temporary variable eliminates the possibility of ending up with a partially initialized object, should an exception be thrown during initialization. Instead, the newly constructed object is completely abandoned:

```
Point p = null;
int zero = 0;
try   { p = new Point { X = 3, Y = 4 / zero }; }
        // Throws DivideByZeroException
catch { Console.WriteLine (p == null);         }
        // True
```

ERIC LIPPERT The use of a hidden temporary variable also makes it clear what the definite assignment rules are. The second line here is not equivalent to the first:

```
Point p1 = new Point(); p1.Y = p1.X; // Legal
Point p2 = new Point() { Y = p2.X }; // Not legal; p2 is not assigned yet
```

If `Rectangle`'s constructor allocates the two embedded `Point` instances:

```
public class Rectangle
{
    Point p1 = new Point();
    Point p2 = new Point();
    public Point P1 { get { return p1; } }
    public Point P2 { get { return p2; } }
}
```


www.it-ebooks.info

The following construct can be used to initialize the embedded `Point` instances instead of assigning new instances:

```
Rectangle r = new Rectangle {
    P1 = { X = 0, Y = 1 },
    P2 = { X = 2, Y = 3 }
};
```

which has the same effect as

```
Rectangle __r = new Rectangle();
__r.P1.X = 0;
__r.P1.Y = 1;
__r.P2.X = 2;
__r.P2.Y = 3;
Rectangle r = __r;
```

7.6.10.3 *Collection Initializers*

A collection initializer specifies the elements of a collection.

collection-initializer:
    `{` *element-initializer-list* `}`
    `{` *element-initializer-list* `,` `}`

element-initializer-list:
    element-initializer
    element-initializer-list `,` *element-initializer*

element-initializer:
    non-assignment-expression
    `{` *expression-list* `}`

expression-list:
    expression
    expression-list `,` *expression*

A collection initializer consists of a sequence of element initializers, enclosed by `{` and `}` tokens and separated by commas. Each element initializer specifies an element to be added to the collection object being initialized, and consists of a list of expressions enclosed by `{` and `}` tokens and separated by commas. A single-expression element initializer can be written without braces, but cannot then be an assignment expression, to avoid ambiguity with member initializers. The *non-assignment-expression* production is defined in §7.18.

The following is an example of an object creation expression that includes a collection initializer:

```
List<int> digits = new List<int> { 0, 1, 2, 3, 4, 5, 6, 7, 8, 9 };
```


www.it-ebooks.info

The collection object to which a collection initializer is applied must be of a type that implements `System.Collections.IEnumerable` or a compile-time error occurs. For each specified element in order, the collection initializer invokes an `Add` method on the target object with the expression list of the element initializer as the argument list, applying normal overload resolution for each invocation. Thus the collection object must contain an applicable `Add` method for each element initializer.

> **ERIC LIPPERT** The rule here is a bit odd: A collection initializer is valid only when the object implements `IEnumerable` and has an `Add` method. Notice that we never call any method of `IEnumerable` in a collection initializer! So why do we require it? The C# design team did a survey of existing objects and made the following discoveries. First, almost all objects that have an `Add` method either are collections or are implementing some kind of arithmetic. We explicitly did not want this design to be a syntactic sugar for arithmetic—just collection creation. Second, of those objects that were collections, there was no one common interface with an `Add` method implemented by all of them. Third, all of the collections implemented `IEnumerable`, but none of the arithmetic objects did. For all these reasons, we decided to use the existence of `IEnumerable` plus an `Add` method as our touchstone for whether a collection initializer is valid.

The following class represents a contact with a name and a list of phone numbers:

```
public class Contact
{
    string name;
    List<string> phoneNumbers = new List<string>();

    public string Name { get { return name; } set { name = value; } }

    public List<string> PhoneNumbers { get { return phoneNumbers; } }
}
```

A `List<Contact>` can be created and initialized as follows:

```
var contacts = new List<Contact> {
    new Contact {
        Name = "Chris Smith",
        PhoneNumbers = { "206-555-0101", "425-882-8080" }
    },
    new Contact {
        Name = "Bob Harris",
        PhoneNumbers = { "650-555-0199" }
    }
};
```


www.it-ebooks.info

which has the same effect as

```
var __clist = new List<Contact>();
Contact __c1 = new Contact();
__c1.Name = "Chris Smith";
__c1.PhoneNumbers.Add("206-555-0101");
__c1.PhoneNumbers.Add("425-882-8080");
__clist.Add(__c1);
Contact __c2 = new Contact();
__c2.Name = "Bob Harris";
__c2.PhoneNumbers.Add("650-555-0199");
__clist.Add(__c2);
var contacts = __clist;
```

where `__clist`, `__c1`, and `__c2` are temporary variables that are otherwise invisible and inaccessible.

7.6.10.4 *Array Creation Expressions*

An *array-creation-expression* is used to create a new instance of an *array-type*.

array-creation-expression:
`new` *non-array-type* `[` *expression-list* `]` *rank-specifiers*$_{opt}$ *array-initializer*$_{opt}$
`new` *array-type* *array-initializer*
`new` *rank-specifier* *array-initializer*

An array creation expression of the first form allocates an array instance of the type that results from deleting each of the individual expressions from the expression list. For example, the array creation expression `new int[10, 20]` produces an array instance of type `int[,]`, and the array creation expression `new int[10][,]` produces an array of type `int[][,]`. Each expression in the expression list must be of type `int`, `uint`, `long`, or `ulong`, or implicitly convertible to one or more of these types. The value of each expression determines the length of the corresponding dimension in the newly allocated array instance. Since the length of an array dimension must be non-negative, it is a compile-time error to have a *constant-expression* with a negative value in the expression list.

Except in an unsafe context (§18.1), the layout of arrays is unspecified.

If an array creation expression of the first form includes an array initializer, each expression in the expression list must be a constant, and the rank and dimension lengths specified by the expression list must match those of the array initializer.

In an array creation expression of the second or third form, the rank of the specified array type or rank specifier must match that of the array initializer. The individual dimension

www.it-ebooks.info

lengths are inferred from the number of elements in each of the corresponding nesting levels of the array initializer. Thus the expression

```
new int[,] {{0, 1}, {2, 3}, {4, 5}}
```

exactly corresponds to

```
new int[3, 2] {{0, 1}, {2, 3}, {4, 5}}
```

An array creation expression of the third form is referred to as an ***implicitly typed array creation expression***. It is similar to the second form, except that the element type of the array is not explicitly given, but determined as the best common type (§7.5.2.14) of the set of expressions in the array initializer. For a multidimensional array—that is, one where the *rank-specifier* contains at least one comma—this set comprises all *expression*s found in nested *array-initializer*s.

Array initializers are described further in §12.6.

The result of evaluating an array creation expression is classified as a value—namely, a reference to the newly allocated array instance. The runtime processing of an array creation expression consists of the following steps:

- The dimension length expressions of the *expression-list* are evaluated in order, from left to right. Following evaluation of each expression, an implicit conversion (§6.1) to one of the following types is performed: `int`, `uint`, `long`, `ulong`. The first type in this list for which an implicit conversion exists is chosen. If evaluation of an expression or the subsequent implicit conversion causes an exception, then no further expressions are evaluated and no further steps are executed.
- The computed values for the dimension lengths are validated as follows. If one or more of the values are less than zero, a `System.OverflowException` is thrown and no further steps are executed.
- An array instance with the given dimension lengths is allocated. If there is not enough memory available to allocate the new instance, a `System.OutOfMemoryException` is thrown and no further steps are executed.
- All elements of the new array instance are initialized to their default values (§5.2).
- If the array creation expression contains an array initializer, then each expression in the array initializer is evaluated and assigned to its corresponding array element. The evaluations and assignments are performed in the order the expressions are written in the array initializer—in other words, elements are initialized in increasing index order, with the rightmost dimension increasing first. If evaluation of a given expression or the subsequent assignment to the corresponding array element causes an exception, then no further elements are initialized (and the remaining elements will thus have their default values).

www.it-ebooks.info

An array creation expression permits instantiation of an array with elements of an array type, but the elements of such an array must be manually initialized. For example, the statement

```
int[][] a = new int[100][];
```

creates a single-dimensional array with 100 elements of type `int[]`. The initial value of each element is `null`. It is not possible for the same array creation expression to also instantiate the subarrays, and the statement

```
int[][] a = new int[100][5];        // Error
```

results in a compile-time error. Instantiation of the subarrays must instead be performed manually, as in

```
int[][] a = new int[100][];
for (int i = 0; i < 100; i++) a[i] = new int[5];
```

BILL WAGNER The comment below violates an FxCop rule, but does explain under which conditions you should violate that rule.

When an array of arrays has a "rectangular" shape—that is, when the subarrays are all of the same length—it is more efficient to use a multi-dimensional array. In the example above, instantiation of the array of arrays creates 101 objects—one outer array and 100 subarrays. In contrast,

```
int[,] = new int[100, 5];
```

creates only a single object—that is, a two-dimensional array—and accomplishes the allocation in a single statement.

PETER SESTOFT It is somewhat dubious for a language specification to contain statements about efficiency, which is a matter dealing with implementation, not semantics. Indeed, whereas it may be faster to allocate (and subsequently manage) a single block of 500 integers than 100 blocks of 5 integers, a matrix multiplication algorithm that works on the "slow" array-of-arrays representation may be faster than one that works on the "efficient" rectangular array representation—probably thanks to a combination of JIT-compiler cleverness and modern CPU architecture.

The following are examples of implicitly typed array creation expressions:

```
var a = new[] { 1, 10, 100, 1000 };                    // int[]
var b = new[] { 1, 1.5, 2, 2.5 };                      // double[]
var c = new[,] { { "hello", null }, { "world", "!" } }; // string[,]
var d = new[] { 1, "one", 2, "two" };                  // Error
```


www.it-ebooks.info

The last expression causes a compile-time error because neither `int` nor `string` is implicitly convertible to the other, so there is no best common type. An explicitly typed array creation expression must be used in this case—for example, specifying the type to be `object[]`. Alternatively, one of the elements can be cast to a common base type, which would then become the inferred element type.

Implicitly typed array creation expressions can be combined with anonymous object initializers (§7.6.10.6) to create anonymously typed data structures. For example:

```
var contacts = new[] {
    new {
        Name = "Chris Smith",
        PhoneNumbers = new[] { "206-555-0101", "425-882-8080" }
    },
    new {
        Name = "Bob Harris",
        PhoneNumbers = new[] { "650-555-0199" }
    }
};
```

7.6.10.5 *Delegate Creation Expressions*

A *delegate-creation-expression* is used to create a new instance of a *delegate-type*.

delegate-creation-expression:
`new` *delegate-type* `(` *expression* `)`

The argument of a delegate creation expression must be a method group, an anonymous function, or a value of either the compile-time type `dynamic` or a *delegate-type*. If the argument is a method group, it identifies the method and, for an instance method, the object for which to create a delegate. If the argument is an anonymous function, it directly defines the parameters and method body of the delegate target. If the argument is a value, it identifies a delegate instance of which to create a copy.

If the *expression* has the compile-time type `dynamic`, the *delegate-creation-expression* is dynamically bound (§7.2.2), and the rules below are applied at runtime using the runtime type of the *expression*. Otherwise, the rules are applied at compile time.

The binding-time processing of a *delegate-creation-expression* of the form `new D(E)`, where `D` is a *delegate-type* and `E` is an *expression*, consists of the following steps:

- If `E` is a method group, the delegate creation expression is processed in the same way as a method group conversion (§6.6) from `E` to `D`.
- If `E` is an anonymous function, the delegate creation expression is processed in the same way as an anonymous function conversion (§6.5) from `E` to `D`.

www.it-ebooks.info

- If `E` is a value, `E` must be compatible (§15.1) with `D`, and the result is a reference to a newly created delegate of type `D` that refers to the same invocation list as `E`. If `E` is not compatible with `D`, a compile-time error occurs.

The runtime processing of a *delegate-creation-expression* of the form `new D(E)`, where `D` is a *delegate-type* and `E` is an *expression*, consists of the following steps:

- If `E` is a method group, the delegate creation expression is evaluated as a method group conversion (§6.6) from `E` to `D`.
- If `E` is an anonymous function, the delegate creation is evaluated as an anonymous function conversion from `E` to `D` (§6.5).
- If `E` is a value of a *delegate-type*:
  - `E` is evaluated. If this evaluation causes an exception, no further steps are executed.
  - If the value of `E` is `null`, a `System.NullReferenceException` is thrown and no further steps are executed.
  - A new instance of the delegate type `D` is allocated. If there is not enough memory available to allocate the new instance, a `System.OutOfMemoryException` is thrown and no further steps are executed.
  - The new delegate instance is initialized with the same invocation list as the delegate instance given by `E`.

The invocation list of a delegate is determined when the delegate is instantiated and then remains constant for the entire lifetime of the delegate. In other words, it is not possible to change the target callable entities of a delegate once it has been created. When two delegates are combined or one is removed from another (§15.1), a new delegate results; no existing delegate has its contents changed.

It is not possible to create a delegate that refers to a property, indexer, user-defined operator, instance constructor, destructor, or static constructor.

As described above, when a delegate is created from a method group, the formal parameter list and return type of the delegate determine which of the overloaded methods to select. In the example

```
delegate double DoubleFunc(double x);

class A
{
    DoubleFunc f = new DoubleFunc(Square);

    static float Square(float x) {
        return x * x;
    }
```


www.it-ebooks.info

```
    static double Square(double x) {
        return x * x;
    }
}
```

the `A.f` field is initialized with a delegate that refers to the second `Square` method because that method exactly matches the formal parameter list and return type of `DoubleFunc`. Had the second `Square` method not been present, a compile-time error would have occurred.

7.6.10.6 *Anonymous Object Creation Expressions*

An *anonymous-object-creation-expression* is used to create an object of an anonymous type.

anonymous-object-creation-expression:
`new` *anonymous-object-initializer*

anonymous-object-initializer:
`{` *member-declarator-list*$_{opt}$ `}`
`{` *member-declarator-list* `,` `}`

member-declarator-list:
member-declarator
member-declarator-list `,` *member-declarator*

member-declarator:
simple-name
member-access
base-access
identifier `=` *expression*

VLADIMIR RESHETNIKOV A *base-access* can be a *member-declarator* only if it is of the form `base.`*identifier*; that is, no base indexer access is allowed here.

An anonymous object initializer declares an anonymous type and returns an instance of that type. An anonymous type is a nameless class type that inherits directly from `object`. The members of an anonymous type are a sequence of read-only properties inferred from the anonymous object initializer used to create an instance of the type. Specifically, an anonymous object initializer of the form

`new { `p_1 `= `e_1 `, `p_2 `= `e_2 `, ... `p_n `= `e_n `}`

www.it-ebooks.info

declares an anonymous type of the form

```
class __Anonymous1
{
    private readonly T1 f1 ;
    private readonly T2 f2 ;
    ...
    private readonly Tn fn ;

    public __Anonymous1(T1 a1, T2 a2,..., Tn an) {
        f1 = a1 ;
        f2 = a2 ;
        ...
        fn = an ;
    }

    public T1 p1 { get { return f1 ; } }
    public T2 p2 { get { return f2 ; } }
    ...
    public Tn pn { get { return fn ; } }

    public override bool Equals(object o) { ... }
    public override int GetHashCode() { ... }
}
```

where each T_x is the type of the corresponding expression e_x. The expression used in a *member-declarator* must have a type. Thus it is a compile-time error for an expression in a *member-declarator* to be null or an anonymous function. It is also a compile-time error for the expression to have an unsafe type.

ERIC LIPPERT The actual code generated by the Microsoft implementation for an anonymous type is somewhat more complex than this discussion suggests because of the desire (mentioned later) to have structurally equivalent anonymous types be represented by the same type throughout a program. Because the field types could be protected nested types, it becomes difficult to figure out where exactly to generate the anonymous class so that it can be effectively shared among different derived types. For this reason, the Microsoft implementation actually generates a generic class and then constructs it with the appropriate type arguments.

The name of an anonymous type is automatically generated by the compiler and cannot be referenced in program text.

JON SKEET Oxymoronic as it sounds, I would dearly love to see named anonymous types: classes that are as easy to express as anonymous types and that possess the same properties of immutability, natural equality, and type and name safety of properties— but with a name. Although many tools can "expand" anonymous types into equivalent normal classes, the concise expression of the type is lost at that point.

www.it-ebooks.info

Within the same program, two anonymous object initializers that specify a sequence of properties of the same names and compile-time types in the same order will produce instances of the same anonymous type.

In the example

```
var p1 = new { Name = "Lawnmower", Price = 495.00 };
var p2 = new { Name = "Shovel", Price = 26.95 };
p1 = p2;
```

the assignment on the last line is permitted because `p1` and `p2` are of the same anonymous type.

The `Equals` and `GetHashcode` methods on anonymous types override the methods inherited from `object`, and are defined in terms of the `Equals` and `GetHashcode` of the properties, so that two instances of the same anonymous type are equal if and only if all their properties are equal.

A member declarator can be abbreviated to a simple name (§7.5.2), a member access (§7.5.4), or a base access (§7.6.8). This is called a ***projection initializer*** and is shorthand for a declaration of and assignment to a property with the same name. Specifically, member declarators of the forms

identifier *expr* **.** *identifier*

are precisely equivalent to the following, respectively:

identifier **=** *identifier* *identifier* **=** *expr* **.** *identifier*

Thus, in a projection initializer, the *identifier* selects both the value and the field or property to which the value is assigned. Intuitively, a projection initializer projects not just a value, but also the name of the value.

7.6.11 The typeof Operator

The `typeof` operator is used to obtain the `System.Type` object for a type.

typeof-expression:
`typeof` `(` *type* `)`
`typeof` `(` *unbound-type-name* `)`
`typeof` `(` `void` `)`

unbound-type-name:
identifier *generic-dimension-specifier*$_{opt}$
identifier `::` *identifier* *generic-dimension-specifier*$_{opt}$
unbound-type-name **.** *identifier* *generic-dimension-specifier*$_{opt}$

www.it-ebooks.info

generic-dimension-specifier:
    `<` *commas*$_{opt}$ `>`

commas:
    `,`
    commas `,`

The first form of *typeof-expression* consists of a `typeof` keyword followed by a parenthesized *type*. The result of an expression of this form is the `System.Type` object for the indicated type. There is only one `System.Type` object for any given type. This means that for a type `T`, `typeof(T) == typeof(T)` is always true. The *type* cannot be `dynamic`.

The second form of *typeof-expression* consists of a `typeof` keyword followed by a parenthesized *unbound-type-name*. An *unbound-type-name* is very similar to a *type-name* (§3.8) except that an *unbound-type-name* contains *generic-dimension-specifier*s whereas a *type-name* contains *type-argument-list*s. When the operand of a *typeof-expression* is a sequence of tokens that satisfies the grammars of both *unbound-type-name* and *type-name*—namely, when it contains neither a *generic-dimension-specifier* nor a *type-argument-list*—the sequence of tokens is considered to be a *type-name*. The meaning of an *unbound-type-name* is determined as follows:

- Convert the sequence of tokens to a *type-name* by replacing each *generic-dimension-specifier* with a *type-argument-list* having the same number of commas and the keyword `object` as each *type-argument*.
- Evaluate the resulting *type-name*, while ignoring all type parameter constraints.
- The *unbound-type-name* resolves to the unbound generic type associated with the resulting constructed type (§4.4.3).

The result of the *typeof-expression* is the `System.Type` object for the resulting unbound generic type.

The third form of *typeof-expression* consists of a `typeof` keyword followed by a parenthesized `void` keyword. The result of an expression of this form is the `System.Type` object that represents the absence of a type. The type object returned by `typeof(void)` is distinct from the type object returned for any type. This special type object is useful in class libraries that allow reflection onto methods in the language, where those methods wish to have a way to represent the return type of any method, including void methods, with an instance of `System.Type`.

The `typeof` operator can be used on a type parameter. The result is the `System.Type` object for the runtime type that was bound to the type parameter. The `typeof` operator can also be used on a constructed type or an unbound generic type (§4.4.3). The `System.Type` object for an unbound generic type is not the same as the `System.Type` object of the instance type.

www.it-ebooks.info

The instance type is always a closed constructed type at runtime, so its `System.Type` object depends on the runtime type arguments in use, while the unbound generic type has no type arguments.

The example

```
using System;

class X<T>
{
    public static void PrintTypes() {
        Type[] t = {
            typeof(int),
            typeof(System.Int32),
            typeof(string),
            typeof(double[]),
            typeof(void),
            typeof(T),
            typeof(X<T>),
            typeof(X<X<T>>),
            typeof(X<>)
        };
        for (int i = 0; i < t.Length; i++) {
            Console.WriteLine(t[i]);
        }
    }
}

class Test
{
    static void Main() {
        X<int>.PrintTypes();
    }
}
```

produces the following output:

```
System.Int32
System.Int32
System.String
System.Double[]
System.Void
System.Int32
X`1[System.Int32]
X`1[X`1[System.Int32]]
X`1[T]
```

Note that `int` and `System.Int32` are the same type.

Also note that the result of `typeof(X<>)` does not depend on the type argument, but the result of `typeof(X<T>)` does.

www.it-ebooks.info

7.6.12 The checked and unchecked Operators

The `checked` and `unchecked` operators are used to control the ***overflow checking context*** for integral-type arithmetic operations and conversions.

checked-expression:
    `checked` `(` *expression* `)`

unchecked-expression:
    `unchecked` `(` *expression* `)`

The `checked` operator evaluates the contained expression in a checked context, and the `unchecked` operator evaluates the contained expression in an unchecked context. A *checked-expression* or *unchecked-expression* corresponds exactly to a *parenthesized-expression* (§7.6.3), except that the contained expression is evaluated in the given overflow checking context.

> **JON SKEET** In most cases, overflow indicates an error—but in my experience, most developers (including myself) usually build without the checking feature on by default. Obviously, this approach has a performance penalty, but that's usually preferable to silently corrupted data. It would possibly make sense to make `/checked+` the default for debug builds and `/checked-` the default for release builds—although I'm generally wary of behavioral changes between debug and release. This would be a little like the behavior of cross-thread exceptions in Windows Forms.
>
> One situation where `unchecked` overflow behavior is almost always appropriate is in `GetHashCode` implementations, where the magnitude of the value generated doesn't really matter; in essence, it's just an arbitrary bit pattern.

The overflow checking context can also be controlled through the `checked` and `unchecked` statements (§8.11).

The following operations are affected by the overflow checking context established by the `checked` and `unchecked` operators and statements:

- The predefined `++` and `--` unary operators (§7.6.9 and §7.7.5), when the operand is of an integral type.
- The predefined `-` unary operator (§7.7.2), when the operand is of an integral type.
- The predefined `+`, `-`, `*`, and `/` binary operators (§7.8), when both operands are of integral types.

www.it-ebooks.info

- Explicit numeric conversions (§6.2.1) from one integral type to another integral type, or from `float` or `double` to an integral type.

When one of the above operations produces a result that is too large to represent in the destination type, the context in which the operation is performed controls the resulting behavior:

- In a `checked` context, if the operation is a constant expression (§7.19), a compile-time error occurs. Otherwise, when the operation is performed at runtime, a `System.OverflowException` is thrown.
- In an `unchecked` context, the result is truncated by discarding any high-order bits that do not fit in the destination type.

For nonconstant expressions (expressions that are evaluated at runtime) that are not enclosed by any `checked` or `unchecked` operators or statements, the default overflow checking context is `unchecked` unless external factors (such as compiler switches and execution environment configuration) call for `checked` evaluation.

For constant expressions (expressions that can be fully evaluated at compile time), the default overflow checking context is always `checked`. Unless a constant expression is explicitly placed in an `unchecked` context, overflows that occur during the compile-time evaluation of the expression always cause compile-time errors.

The body of an anonymous function is not affected by `checked` or `unchecked` contexts in which the anonymous function occurs.

In the example

```
class Test
{
    static readonly int x = 1000000;
    static readonly int y = 1000000;

    static int F() {
        return checked(x * y);  // Throws OverflowException
    }

    static int G() {
        return unchecked(x * y);    // Returns -727379968
    }

    static int H() {
        return x * y;             // Depends on default
    }
}
```

no compile-time errors are reported since neither of the expressions can be evaluated at compile time. At runtime, the `F` method throws a `System.OverflowException` and the `G`

www.it-ebooks.info

method returns –727379968 (the lower 32 bits of the out-of-range result). The behavior of the `H` method depends on the default overflow checking context for the compilation, but it is either the same as `F` or the same as `G`.

In the example

```
class Test
{
    const int x = 1000000;
    const int y = 1000000;

    static int F() {
        return checked(x * y);      // Compile error, overflow
    }

    static int G() {
        return unchecked(x * y);    // Returns -727379968
    }

    static int H() {
        return x * y;             // Compile error, overflow
    }
}
```

the overflows that occur when evaluating the constant expressions in `F` and `H` cause compile-time errors to be reported because the expressions are evaluated in a `checked` context. An overflow also occurs when evaluating the constant expression in `G`, but since the evaluation takes place in an `unchecked` context, the overflow is not reported.

The `checked` and `unchecked` operators affect the overflow checking context only for those operations that are textually contained within the "`(`" and "`)`" tokens. The operators have no effect on function members that are invoked as a result of evaluating the contained expression. In the example

```
class Test
{
    static int Multiply(int x, int y) {
        return x * y;
    }

    static int F() {
        return checked(Multiply(1000000, 1000000));
    }
}
```

the use of `checked` in `F` does not affect the evaluation of `x * y` in `Multiply`, so `x * y` is evaluated in the default overflow checking context.

The `unchecked` operator is convenient when writing constants of the signed integral types in hexadecimal notation. For example:

www.it-ebooks.info

```
class Test
{
    public const int AllBits = unchecked((int)0xFFFFFFFF);

    public const int HighBit = unchecked((int)0x80000000);
}
```

Both of the hexadecimal constants above are of type `uint`. Because the constants are outside the `int` range, without the `unchecked` operator, the casts to `int` would produce compile-time errors.

The `checked` and `unchecked` operators and statements allow programmers to control certain aspects of some numeric calculations. However, the behavior of some numeric operators depends on their operands' data types. For example, multiplying two decimals always results in an exception on overflow *even* within an explicitly `unchecked` construct. Similarly, multiplying two floats never results in an exception on overflow *even* within an explicitly `checked` construct. In addition, other operators are *never* affected by the mode of checking, whether default or explicit.

7.6.13 Default Value Expressions

A default value expression is used to obtain the default value (§5.2) of a type. Typically a default value expression is used for type parameters, since it may not be known if the type parameter is a value type or a reference type. (No conversion exists from the `null` literal to a type parameter unless the type parameter is known to be a reference type.)

default-value-expression:
    `default` (*type*)

If the *type* in a *default-value-expression* evaluates at runtime to a reference type, the result is `null` converted to that type. If the *type* in a *default-value-expression* evaluates at runtime to a value type, the result is the *value-type*'s default value (§4.1.2).

A *default-value-expression* is a constant expression (§7.19) if the type is a reference type or a type parameter that is known to be a reference type (§10.1.5). In addition, a *default-value-expression* is a constant expression if the type is one of the following value types: `sbyte`, `byte`, `short`, `ushort`, `int`, `uint`, `long`, `ulong`, `char`, `float`, `double`, `decimal`, `bool`, or any enumeration type.

JOSEPH ALBAHARI A default value expression is particularly useful with generic type parameters. Microsoft's LINQ implementation makes extensive use of this construct, in operators such as `FirstOrDefault`, `SingleOrDefault`, and `DefaultIfEmpty`.

www.it-ebooks.info

7.6.14 Anonymous Method Expressions

An *anonymous-method-expression* is one of two ways of defining an anonymous function. These are further described in §7.15.

7.7 Unary Operators

The `+`, `-`, `!`, `~`, `++`, `--`, and cast operators are called the unary operators.

unary-expression:
primary-expression
`+` *unary-expression*
`-` *unary-expression*
`!` *unary-expression*
`~` *unary-expression*
pre-increment-expression
pre-decrement-expression
cast-expression

If the operand of a *unary-expression* has the compile-time type `dynamic`, it is dynamically bound (§7.2.2). In this case, the compile-time type of the *unary-expression* is `dynamic`, and the resolution described below will take place at runtime using the runtime type of the operand.

7.7.1 Unary Plus Operator

For an operation of the form `+x`, unary operator overload resolution (§7.3.3) is applied to select a specific operator implementation. The operand is converted to the parameter type of the selected operator, and the type of the result is the return type of the operator. The predefined unary plus operators are listed here:

```
int operator +(int x);
uint operator +(uint x);
long operator +(long x);
ulong operator +(ulong x);
float operator +(float x);
double operator +(double x);
decimal operator +(decimal x);
```

For each of these operators, the result is simply the value of the operand.

ERIC LIPPERT The unary plus operator is the world's least useful operator. It is included for completeness, so that you can write `int x = -30; int y = +40;` should you wish to emphasize that the value is positive for readability purposes.

www.it-ebooks.info

7.7.2 Unary Minus Operator

For an operation of the form `-x`, unary operator overload resolution (§7.3.3) is applied to select a specific operator implementation. The operand is converted to the parameter type of the selected operator, and the type of the result is the return type of the operator. The predefined negation operators are identified here:

- Integer negation:

```
int operator -(int x);
long operator -(long x);
```

  The result is computed by subtracting `x` from zero. If the value of `x` is the smallest representable value of the operand type (-2^{31} for `int` or -2^{63} for `long`), then the mathematical negation of `x` is not representable within the operand type. If this occurs within a `checked` context, a `System.OverflowException` is thrown; if it occurs within an `unchecked` context, the result is the value of the operand and the overflow is not reported.

  If the operand of the negation operator is of type `uint`, it is converted to type `long`, and the type of the result is `long`. An exception is the rule that permits the `int` value –2147483648 (-2^{31}) to be written as a decimal integer literal (§2.4.4.2).

  If the operand of the negation operator is of type `ulong`, a compile-time error occurs. An exception is the rule that permits the `long` value –9223372036854775808 (-2^{63}) to be written as a decimal integer literal (§2.4.4.2).

- Floating point negation:

```
float operator -(float x);
double operator -(double x);
```

  The result is the value of `x` with its sign inverted. If `x` is NaN, the result is also NaN.

- Decimal negation:

```
decimal operator -(decimal x);
```

  The result is computed by subtracting `x` from zero. Decimal negation is equivalent to using the unary minus operator of type `System.Decimal`.

7.7.3 Logical Negation Operator

For an operation of the form `!x`, unary operator overload resolution (§7.3.3) is applied to select a specific operator implementation. The operand is converted to the parameter type of the selected operator, and the type of the result is the return type of the operator. Only one predefined logical negation operator exists:

```
bool operator !(bool x);
```


www.it-ebooks.info

This operator computes the logical negation of the operand: If the operand is `true`, the result is `false`. If the operand is `false`, the result is `true`.

7.7.4 Bitwise Complement Operator

For an operation of the form `~x`, unary operator overload resolution (§7.3.3) is applied to select a specific operator implementation. The operand is converted to the parameter type of the selected operator, and the type of the result is the return type of the operator. The predefined bitwise complement operators are listed here:

```
int operator ~(int x);
uint operator ~(uint x);
long operator ~(long x);
ulong operator ~(ulong x);
```

For each of these operators, the result of the operation is the bitwise complement of `x`.

Every enumeration type `E` implicitly provides the following bitwise complement operator:

```
E operator ~(E x);
```

> **BILL WAGNER** This operator will often create an invalid value for the enumeration.

The result of evaluating `~x`, where `x` is an expression of an enumeration type `E` with an underlying type `U`, is exactly the same as evaluating `(E)(~(U)x)`.

7.7.5 Prefix Increment and Decrement Operators

pre-increment-expression:
`++` *unary-expression*

pre-decrement-expression:
`--` *unary-expression*

The operand of a prefix increment or decrement operation must be an expression classified as a variable, a property access, or an indexer access. The result of the operation is a value of the same type as the operand.

If the operand of a prefix increment or decrement operation is a property or indexer access, the property or indexer must have both a `get` and a `set` accessor. If this is not the case, a binding-time error occurs.

www.it-ebooks.info

Unary operator overload resolution (§7.3.3) is applied to select a specific operator implementation. Predefined `++` and `--` operators exist for the following types: `sbyte`, `byte`, `short`, `ushort`, `int`, `uint`, `long`, `ulong`, `char`, `float`, `double`, `decimal`, and any enum type. The predefined `++` operators return the value produced by adding 1 to the operand, and the predefined `--` operators return the value produced by subtracting 1 from the operand. In a `checked` context, if the result of this addition or subtraction is outside the range of the result type and the result type is an integral type or enum type, a `System.OverflowException` is thrown.

The runtime processing of a prefix increment or decrement operation of the form `++x` or `--x` consists of the following steps:

- If `x` is classified as a variable:
  - `x` is evaluated to produce the variable.
  - The selected operator is invoked with the value of `x` as its argument.
  - The value returned by the operator is stored in the location given by the evaluation of `x`.
  - The value returned by the operator becomes the result of the operation.
- If `x` is classified as a property or indexer access:
  - The instance expression (if `x` is not `static`) and the argument list (if `x` is an indexer access) associated with `x` are evaluated, and the results are used in the subsequent `get` and `set` accessor invocations.
  - The `get` accessor of `x` is invoked.
  - The selected operator is invoked with the value returned by the `get` accessor as its argument.
  - The `set` accessor of `x` is invoked with the value returned by the operator as its `value` argument.

VLADIMIR RESHETNIKOV If either the *get-accessor* or *set-accessor* is missing or is not accessible, a compile-time error occurs.

  - The value returned by the operator becomes the result of the operation.

The `++` and `--` operators also support postfix notation (§7.6.9). Typically, the result of `x++` or `x--` is the value of `x` *before* the operation, whereas the result of `++x` or `--x` is the value of `x` *after* the operation. In either case, `x` itself has the same value after the operation.

www.it-ebooks.info

ERIC LIPPERT Note the weasel word "typically" in the specification. Indeed, this is almost always the case, but the specification does not *require* that to be the case. The value of `x` after the operation could be any old thing; someone could be mutating `x` on another thread, `x` could be a property that returns random values when you call its getter, and so on.

Consider this code:

```
index = 0; value = this.arr[index++];
```

I often hear this code described as follows: "This gets the element at index zero and then increments index to one; the increment happens after the array lookup because the `++` comes after the index." If you read the specification carefully, you'll see that this statement is completely wrong. The correct order of events is (1) remember the current value of index, (2) increment `index`, and (3) do the array lookup using the remembered value. The increment happens *before* the lookup, not *after* it.

An `operator ++` or `operator --` implementation can be invoked using either postfix or prefix notation. It is not possible to have separate operator implementations for the two notations.

7.7.6 Cast Expressions

A *cast-expression* is used to explicitly convert an expression to a given type.

cast-expression:
`(` *type* `)` *unary-expression*

A *cast-expression* of the form `(T)E`, where `T` is a *type* and `E` is a *unary-expression*, performs an explicit conversion (§6.2) of the value of `E` to type `T`. If no explicit conversion exists from `E` to `T`, a binding-time error occurs. Otherwise, the result is the value produced by the explicit conversion. The result is always classified as a value, even if `E` denotes a variable.

ERIC LIPPERT The cast operator is a bit of an odd duck, both because of its unusual syntax and because of its semantics. A cast operator is usually used to mean either (1) I claim that at runtime the value will *not* be of the cast type; do whatever it takes to make me a new value of the required type; or (2) I claim that at runtime the value *will* be of the cast type; verify my claim by throwing an exception if I'm wrong. The attentive reader will note that these are *opposites.* Though it's a neat trick to have one operator do two opposite things, as a result, it's easy to get confused about what the cast operator is *actually* doing.

www.it-ebooks.info

The grammar for a *cast-expression* leads to certain syntactic ambiguities. For example, the expression `(x)-y` could be interpreted either as a *cast-expression* (a cast of `-y` to type `x`) or as an *additive-expression* combined with a *parenthesized-expression* (which computes the value `x - y`).

To resolve *cast-expression* ambiguities, the following rule exists: A sequence of one or more *token*s (§2.3.3) enclosed in parentheses is considered the start of a *cast-expression* only if at least one of the following conditions is true:

- The sequence of tokens is correct grammar for a *type*, but not for an *expression*.
- The sequence of tokens is correct grammar for a *type*, and the token immediately following the closing parentheses is the token "`~`", the token "`!`", the token "`(`", an *identifier* (§2.4.1), a *literal* (§2.4.4), or any *keyword* (§2.4.3) except `as` and `is`.

The term "correct grammar" above means only that the sequence of tokens must conform to the particular grammatical production. It specifically does not consider the actual meaning of any constituent identifiers. For example, if `x` and `y` are identifiers, then `x.y` is correct grammar for a type, even if `x.y` doesn't actually denote a type.

From the disambiguation rule it follows that, if `x` and `y` are identifiers, `(x)y`, `(x)(y)`, and `(x)(-y)` are *cast-expression*s, but `(x)-y` is not, even if `x` identifies a type. However, if `x` is a keyword that identifies a predefined type (such as `int`), then all four forms are *cast-expression*s (because such a keyword could not possibly be an expression by itself).

7.8 Arithmetic Operators

The `*`, `/`, `%`, `+`, and `-` operators are called the arithmetic operators.

multiplicative-expression:
    unary-expression
    multiplicative-expression `*` *unary-expression*
    multiplicative-expression `/` *unary-expression*
    multiplicative-expression `%` *unary-expression*

additive-expression:
    multiplicative-expression
    additive-expression `+` *multiplicative-expression*
    additive-expression `-` *multiplicative-expression*

If an operand of an arithmetic operator has the compile-time type `dynamic`, then the expression is dynamically bound (§7.2.2). In this case the compile-time type of the expression is `dynamic`, and the resolution described below will take place at runtime using the runtime type of those operands that have the compile-time type `dynamic`.

www.it-ebooks.info

PETER SESTOFT When a value of a simple type such as `double` is assigned to a variable of type `dynamic`, it must typically be boxed (stored as an object in the heap) at runtime. Thus arithmetic computations with operands of type `dynamic` are likely to be slower than those with simple compile-time types. For instance, this loop is roughly six times slower (on Microsoft .NET 4.0) than if `sum` were declared to have type `double`:

```
dynamic sum = 0;
for (int i=0; i<count; i++)
  sum += (i + 1.0) * i;
```

Actually, this is very fast compared to arithmetics in some other dynamically typed languages.

7.8.1 Multiplication Operator

For an operation of the form `x * y`, binary operator overload resolution (§7.3.4) is applied to select a specific operator implementation. The operands are converted to the parameter types of the selected operator, and the type of the result is the return type of the operator.

The predefined multiplication operators are listed below. The operators all compute the product of `x` and `y`.

- Integer multiplication:

  ```
  int operator *(int x, int y);
  uint operator *(uint x, uint y);
  long operator *(long x, long y);
  ulong operator *(ulong x, ulong y);
  ```

  In a `checked` context, if the product is outside the range of the result type, a `System.OverflowException` is thrown. In an `unchecked` context, overflows are not reported and any significant high-order bits outside the range of the result type are discarded.

- Floating point multiplication:

  ```
  float operator *(float x, float y);
  double operator *(double x, double y);
  ```

  The product is computed according to the rules of IEEE 754 arithmetic. The following table lists the results of all possible combinations of non-zero finite values, zeros, infinities, and NaNs. In the table, `x` and `y` are positive finite values; `z` is the result of `x * y`. If the result is too large for the destination type, `z` is infinity. If the result is too small for the destination type, `z` is zero.

www.it-ebooks.info

	+y	−y	+0	−0	+∞	−∞	NaN
+x	+z	−z	+0	−0	+∞	−∞	NaN
−x	−z	+z	−0	+0	−∞	+∞	NaN
+0	+0	−0	+0	−0	NaN	NaN	NaN
−0	−0	+0	−0	+0	NaN	NaN	NaN
+∞	+∞	−∞	NaN	NaN	+∞	−∞	NaN
−∞	−∞	+∞	NaN	NaN	−∞	+∞	NaN
NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN

- Decimal multiplication:

```
decimal operator *(decimal x, decimal y);
```

If the resulting value is too large to represent in the `decimal` format, a `System.Overflow-Exception` is thrown. If the result value is too small to represent in the `decimal` format, the result is zero. The scale of the result, before any rounding, is the sum of the scales of the two operands.

Decimal multiplication is equivalent to using the multiplication operator of type `System.Decimal`.

PETER SESTOFT Where the text says "IEEE 754 floating point" here and elsewhere in this chapter, it would be more precise to say "IEEE 754 binary floating point," because the 2008 version of that IEEE standard describes binary as well as decimal floating point, and those are very different things!

Also, the following rule is prescribed by the IEEE 754 binary floating point standard, and respected by the current Microsoft and Mono C# implementations: Whenever one or more operands is a NaN, the result is a NaN, *and the NaN payload of the result equals the payload of one of the NaN operands.* The NaN payload of a 64-bit `double` consists of its 51 least significant bits (so a `double` can represent 251—roughly 2*101—different NaNs). The NaN payload of a 32-bit `float` consists of its 22 least significant bits (so a `float` can represent 222—roughly 4*106—different NaNs).

Continued

www.it-ebooks.info

The NaN payload preservation is important for efficient scientific computing. For this approach to really work, in addition to the arithmetic operators, the mathematical functions in `System.Math` should respect IEEE binary floating point as well. In the current implementations, they mostly do so.

The .NET library provides a method `System.Double.IsNan(d)` to test whether a `double` is a NaN.

For type `double`, it also provides methods `DoubleToInt64Bits` and `Int64BitsToDouble` that can be used to get and set NaN payload bits:

```
public static long GetNanPayload(double d) {
    return System.BitConverter.DoubleToInt64Bits(d) & 0x0007FFFFFFFFFFFF;
}

public static double MakeNanPayload(long nanbits) {
    nanbits &= 0x0007FFFFFFFFFFFF;
    nanbits |= System.BitConverter.DoubleToInt64Bits(Double.NaN);
    return System.BitConverter.Int64BitsToDouble(nanbits);
}
```

Strangely, the .NET library has no corresponding methods for type `float`, but one can use unsafe pointer conversions to achieve the same effect. For the gory details, see an annotation on §18.4.

7.8.2 Division Operator

For an operation of the form `x / y`, binary operator overload resolution (§7.3.4) is applied to select a specific operator implementation. The operands are converted to the parameter types of the selected operator, and the type of the result is the return type of the operator.

The predefined division operators are listed below. The operators all compute the quotient of `x` and `y`.

- Integer division:

```
int operator /(int x, int y);
uint operator /(uint x, uint y);
long operator /(long x, long y);
ulong operator /(ulong x, ulong y);
```

  If the value of the right operand is zero, a `System.DivideByZeroException` is thrown.

  The division rounds the result toward zero. Thus the absolute value of the result is the largest possible integer that is less than or equal to the absolute value of the quotient of the two operands. The result is zero or positive when the two operands have the same sign and zero or negative when the two operands have opposite signs.

www.it-ebooks.info

If the left operand is the smallest representable `int` or `long` value and the right operand is `-1`, an overflow occurs. In a `checked` context, this causes a `System.ArithmeticException` (or a subclass thereof) to be thrown. In an `unchecked` context, it is implementation-defined as to whether a `System.ArithmeticException` (or a subclass thereof) is thrown or the overflow goes unreported with the resulting value being that of the left operand.

- Floating point division:

```
float operator /(float x, float y);
double operator /(double x, double y);
```

The quotient is computed according to the rules of IEEE 754 arithmetic. The following table lists the results of all possible combinations of non-zero finite values, zeros, infinities, and NaNs. In the table, `x` and `y` are positive finite values; `z` is the result of `x / y`. If the result is too large for the destination type, `z` is infinity. If the result is too small for the destination type, `z` is zero.

	+y	−y	+0	−0	+∞	−∞	NaN
+x	+z	−z	+∞	−∞	+0	−0	NaN
−x	−z	+z	−∞	+∞	−0	+0	NaN
+0	+0	−0	NaN	NaN	+0	−0	NaN
−0	−0	+0	NaN	NaN	−0	+0	NaN
+∞	+∞	−∞	+∞	−∞	NaN	NaN	NaN
−∞	−∞	+∞	−∞	+∞	NaN	NaN	NaN
NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN

- Decimal division:

```
decimal operator /(decimal x, decimal y);
```

If the value of the right operand is zero, a `System.DivideByZeroException` is thrown. If the resulting value is too large to represent in the `decimal` format, a `System.OverflowException` is thrown. If the result value is too small to represent in the `decimal` format, the result is zero. The scale of the result is the smallest scale that will preserve a result equal to the nearest representable decimal value to the true mathematical result.

Decimal division is equivalent to using the division operator of type `System.Decimal`.

www.it-ebooks.info

7.8.3 Remainder Operator

For an operation of the form `x % y`, binary operator overload resolution (§7.3.4) is applied to select a specific operator implementation. The operands are converted to the parameter types of the selected operator, and the type of the result is the return type of the operator.

The predefined remainder operators are listed below. The operators all compute the remainder of the division between `x` and `y`.

- Integer remainder:

```
int operator %(int x, int y);
uint operator %(uint x, uint y);
long operator %(long x, long y);
ulong operator %(ulong x, ulong y);
```

The result of `x % y` is the value produced by `x - (x / y) * y`. If `y` is zero, a `System.DivideByZeroException` is thrown.

> **ERIC LIPPERT** Consider this code:
>
> ```
> static bool IsOdd(int x) { return x%2 == 1; }
> ```
>
> Is it correct? No! By the sentence above, this is the same as
>
> ```
> static bool IsOdd(int x) { return x-(x/2)*2 == 1; }
> ```
>
> which is false if `x` is `-1`. The correct code is
>
> ```
> static bool IsOdd(int x) { return x%2 != 0; }
> ```

If the left operand is the smallest `int` or `long` value and the right operand is `-1`, a `System.OverflowException` is thrown. In no case does `x % y` throw an exception where `x / y` would not throw an exception.

- Floating point remainder:

```
float operator %(float x, float y);
double operator %(double x, double y);
```

The following table lists the results of all possible combinations of non-zero finite values, zeros, infinities, and NaNs. In the table, `x` and `y` are positive finite values; `z` is the result of `x % y` and is computed as `x - n * y`, where `n` is the largest possible integer that is less than or equal to `x / y`. This method of computing the remainder is analogous to that used for integer operands, but differs from the IEEE 754 definition (in which `n` is the integer closest to `x / y`).

www.it-ebooks.info

	+y	-y	+0	-0	+∞	-∞	NaN
+x	+z	+z	NaN	NaN	x	x	NaN
-x	-z	-z	NaN	NaN	-x	-x	NaN
+0	+0	+0	NaN	NaN	+0	+0	NaN
-0	-0	-0	NaN	NaN	-0	-0	NaN
+∞	NaN	NaN	NaN	NaN	NaN	NaN	NaN
-∞	NaN	NaN	NaN	NaN	NaN	NaN	NaN
NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN

- Decimal remainder:

```
decimal operator %(decimal x, decimal y);
```

  If the value of the right operand is zero, a `System.DivideByZeroException` is thrown. The scale of the result, before any rounding, is the larger of the scales of the two operands, and the sign of the result, if non-zero, is the same as that of `x`.

  Decimal remainder is equivalent to using the remainder operator of type `System.Decimal`.

CHRIS SELLS The "remainder" operator is also known as "modulo" in other computer languages, sometimes denoted as "mod." If you're really a geek, you'll find yourself using "modulo" in (loosely) human sentences instead of the phrase "except for," as in "Modulo testing, debugging, and documentation, we're ready to ship!"

7.8.4 Addition Operator

For an operation of the form `x + y`, binary operator overload resolution (§7.3.4) is applied to select a specific operator implementation. The operands are converted to the parameter types of the selected operator, and the type of the result is the return type of the operator.

The predefined addition operators are listed below. For numeric and enumeration types, the predefined addition operators compute the sum of the two operands. When one or

www.it-ebooks.info

both operands are of type `string`, the predefined addition operators concatenate the string representation of the operands.

- Integer addition:

```
int operator +(int x, int y);
uint operator +(uint x, uint y);
long operator +(long x, long y);
ulong operator +(ulong x, ulong y);
```

  In a `checked` context, if the sum is outside the range of the result type, a `System.OverflowException` is thrown. In an `unchecked` context, overflows are not reported and any significant high-order bits outside the range of the result type are discarded.

- Floating point addition:

```
float operator +(float x, float y);
double operator +(double x, double y);
```

  The sum is computed according to the rules of IEEE 754 arithmetic. The following table lists the results of all possible combinations of non-zero finite values, zeros, infinities, and NaNs. In the table, x and y are non-zero finite values; z is the result of x + y. If x and y have the same magnitude but opposite signs, z is positive zero. If x + y is too large to represent in the destination type, z is an infinity with the same sign as x + y.

	y	+0	−0	+∞	−∞	NaN
x	z	x	x	+∞	−∞	NaN
+0	y	+0	+0	+∞	−∞	NaN
−0	y	+0	−0	+∞	−∞	NaN
+∞	+∞	+∞	+∞	+∞	NaN	NaN
−∞	−∞	−∞	−∞	NaN	−∞	NaN
NaN	NaN	NaN	NaN	NaN	NaN	NaN

- Decimal addition:

```
decimal operator +(decimal x, decimal y);
```

  If the resulting value is too large to represent in the `decimal` format, a `System.OverflowException` is thrown. The scale of the result, before any rounding, is the larger of the scales of the two operands.

www.it-ebooks.info

Decimal addition is equivalent to using the addition operator of type `System.Decimal`.

- Enumeration addition. Every enumeration type implicitly provides the following predefined operators, where `E` is the enum type and `U` is the underlying type of `E`:

```
E operator +(E x, U y);
E operator +(U x, E y);
```

At runtime, these operators are evaluated exactly as `(E)((U)x + (U)y)`.

> **BILL WAGNER** Once again, addition on enumerations may not result in a valid enumeration member. The same is true for all of the arithmetic operators.

- String concatenation:

```
string operator +(string x, string y);
string operator +(string x, object y);
string operator +(object x, string y);
```

These overloads of the binary + operator perform string concatenation. If an operand of string concatenation is `null`, an empty string is substituted. Otherwise, any non-string argument is converted to its string representation by invoking the virtual `ToString` method inherited from type `object`. If `ToString` returns `null`, an empty string is substituted.

```
using System;

class Test
{
    static void Main() {
        string s = null;
        Console.WriteLine("s = >" + s + "<");   // Displays s = ><
        int i = 1;
        Console.WriteLine("i = " + i);          // Displays i = 1
        float f = 1.2300E+15F;
        Console.WriteLine("f = " + f);          // Displays f = 1.23E+15
        decimal d = 2.900m;
        Console.WriteLine("d = " + d);          // Displays d = 2.900
    }
}
```

The result of the string concatenation operator is a string that consists of the characters of the left operand followed by the characters of the right operand. The string concatenation operator never returns a `null` value. A `System.OutOfMemoryException` may be thrown if there is not enough memory available to allocate the resulting string.

www.it-ebooks.info

JON SKEET It's slightly surprising that the .NET `System.String` type doesn't have an overload for `+` that would be used in this situation, until you consider the optimizations available to the compiler because it has visibility of more code. An expression such as `a+b+c+d` doesn't have to result in three intermediate strings being produced: A single call to `String.Concat(a, b, c, d)` allows the concatenation to be performed more efficiently.

- Delegate combination. Every delegate type implicitly provides the following predefined operator, where `D` is the delegate type:

```
D operator +(D x, D y);
```

  The binary `+` operator performs delegate combination when both operands are of some delegate type `D`. (If the operands have different delegate types, a binding-time error occurs.) If the first operand is `null`, the result of the operation is the value of the second operand (even if that is also `null`). Otherwise, if the second operand is `null`, then the result of the operation is the value of the first operand. Otherwise, the result of the operation is a new delegate instance that, when invoked, invokes the first operand and then invokes the second operand. For examples of delegate combination, see §7.8.5 and §15.4. Since `System.Delegate` is not a delegate type, `operator +` is not defined for it.

7.8.5 Subtraction Operator

For an operation of the form `x – y`, binary operator overload resolution (§7.3.4) is applied to select a specific operator implementation. The operands are converted to the parameter types of the selected operator, and the type of the result is the return type of the operator.

The predefined subtraction operators are listed below. The operators all subtract `y` from `x`.

- Integer subtraction:

```
int operator –(int x, int y);
uint operator –(uint x, uint y);
long operator –(long x, long y);
ulong operator –(ulong x, ulong y);
```

  In a `checked` context, if the difference is outside the range of the result type, a `System.OverflowException` is thrown. In an `unchecked` context, overflows are not reported and any significant high-order bits outside the range of the result type are discarded.

- Floating point subtraction:

```
float operator –(float x, float y);
double operator –(double x, double y);
```


www.it-ebooks.info

The difference is computed according to the rules of IEEE 754 arithmetic. The following table lists the results of all possible combinations of non-zero finite values, zeros, infinities, and NaNs. In the table, x and y are non-zero finite values; z is the result of x – y. If x and y are equal, z is positive zero. If x – y is too large to represent in the destination type, z is an infinity with the same sign as x – y.

	y	+0	–0	+∞	–∞	NaN
x	z	x	x	–∞	+∞	NaN
+0	–y	+0	+0	–∞	+∞	NaN
–0	–y	–0	+0	–∞	+∞	NaN
+∞	+∞	+∞	+∞	NaN	+∞	NaN
–∞	–∞	–∞	–∞	–∞	NaN	NaN
NaN	NaN	NaN	NaN	NaN	NaN	NaN

- Decimal subtraction:

```
decimal operator -(decimal x, decimal y);
```

If the resulting value is too large to represent in the `decimal` format, a `System.OverflowException` is thrown. The scale of the result, before any rounding, is the larger of the scales of the two operands.

Decimal subtraction is equivalent to using the subtraction operator of type `System.Decimal`.

- Enumeration subtraction. Every enumeration type implicitly provides the following predefined operator, where `E` is the enum type and `U` is the underlying type of `E`:

```
U operator -(E x, E y);
```

This operator is evaluated exactly as `(U)((U)x - (U)y)`. In other words, the operator computes the difference between the ordinal values of `x` and `y`, and the type of the result is the underlying type of the enumeration.

```
E operator -(E x, U y);
```

This operator is evaluated exactly as `(E)((U)x - y)`. In other words, the operator subtracts a value from the underlying type of the enumeration, yielding a value of the enumeration.

www.it-ebooks.info

- Delegate removal. Every delegate type implicitly provides the following predefined operator, where `D` is the delegate type:

```
D operator –(D x, D y);
```

The binary – operator performs delegate removal when both operands are of some delegate type `D`. If the operands have different delegate types, a binding-time error occurs. If the first operand is `null`, the result of the operation is `null`. Otherwise, if the second operand is `null`, then the result of the operation is the value of the first operand. Otherwise, both operands represent invocation lists (§15.1) having one or more entries, and the result is a new invocation list consisting of the first operand's list with the second operand's entries removed from it, provided the second operand's list is a proper contiguous sublist of the first's. (To determine sublist equality, corresponding entries are compared as for the delegate equality operator (§7.10.8).) Otherwise, the result is the value of the left operand. Neither of the operands' lists is changed in the process. If the second operand's list matches multiple sublists of contiguous entries in the first operand's list, the rightmost matching sublist of contiguous entries is removed. If removal results in an empty list, the result is `null`. For example:

```
delegate void D(int x);

class C
{
    public static void M1(int i) { /* ... */ }
    public static void M2(int i) { /* ... */ }
}

class Test
{
    static void Main() {
        D cd1 = new D(C.M1);
        D cd2 = new D(C.M2);
        D cd3 = cd1 + cd2 + cd2 + cd1; // M1 + M2 + M2 + M1
        cd3 -= cd1;                   // => M1 + M2 + M2

        cd3 = cd1 + cd2 + cd2 + cd1;  // M1 + M2 + M2 + M1
        cd3 -= cd1 + cd2;             // => M2 + M1

        cd3 = cd1 + cd2 + cd2 + cd1;  // M1 + M2 + M2 + M1
        cd3 -= cd2 + cd2;             // => M1 + M1

        cd3 = cd1 + cd2 + cd2 + cd1;  // M1 + M2 + M2 + M1
        cd3 -= cd2 + cd1;             // => M1 + M2

        cd3 = cd1 + cd2 + cd2 + cd1;  // M1 + M2 + M2 + M1
        cd3 -= cd1 + cd1;             // => M1 + M2 + M2 + M1
    }
}
```


www.it-ebooks.info

7.9 Shift Operators

The << and >> operators are used to perform bit shifting operations.

shift-expression:
additive-expression
shift-expression `<<` *additive-expression*
shift-expression *right-shift* *additive-expression*

If an operand of a *shift-expression* has the compile-time type `dynamic`, then the expression is dynamically bound (§7.2.2). In this case, the compile-time type of the expression is `dynamic`, and the resolution described below will take place at runtime using the runtime type of those operands that have the compile-time type `dynamic`.

For an operation of the form `x << count` or `x >> count`, binary operator overload resolution (§7.3.4) is applied to select a specific operator implementation. The operands are converted to the parameter types of the selected operator, and the type of the result is the return type of the operator.

When declaring an overloaded shift operator, the type of the first operand must always be the class or struct containing the operator declaration, and the type of the second operand must always be `int`.

The predefined shift operators are listed below.

- Shift left:

```
int operator <<(int x, int count);
uint operator <<(uint x, int count);
long operator <<(long x, int count);
ulong operator <<(ulong x, int count);
```

  The << operator shifts `x` left by a number of bits computed as described below.

  The high-order bits outside the range of the result type of `x` are discarded, the remaining bits are shifted left, and the low-order empty bit positions are set to zero.

- Shift right:

```
int operator >>(int x, int count);
uint operator >>(uint x, int count);
long operator >>(long x, int count);
ulong operator >>(ulong x, int count);
```

  The >> operator shifts `x` right by a number of bits computed as described below.

www.it-ebooks.info

When x is of type int or long, the low-order bits of x are discarded, the remaining bits are shifted right, and the high-order empty bit positions are set to zero if x is non-negative and set to 1 if x is negative.

When x is of type uint or ulong, the low-order bits of x are discarded, the remaining bits are shifted right, and the high-order empty bit positions are set to zero.

For the predefined operators, the number of bits to shift is computed as follows:

- When the type of x is int or uint, the shift count is given by the low-order five bits of count. In other words, the shift count is computed from count & 0x1F.
- When the type of x is long or ulong, the shift count is given by the low-order six bits of count. In other words, the shift count is computed from count & 0x3F.

If the resulting shift count is zero, the shift operators simply return the value of x.

JON SKEET Masking the shift count may be efficient, but it can make for some very confusing behavior:

```
for (int i = 0; i < 40; i++)
{
    Console.WriteLine(int.MaxValue >> i);
}
```

This code prints values that are halved on each iteration—until they cycle back to int.MaxValue after reaching 0.

Shift operations never cause overflows and produce the same results in checked and unchecked contexts.

When the left operand of the >> operator is of a signed integral type, the operator performs an *arithmetic* shift right wherein the value of the most significant bit (the sign bit) of the operand is propagated to the high-order empty bit positions. When the left operand of the >> operator is of an unsigned integral type, the operator performs a *logical* shift right wherein high-order empty bit positions are always set to zero. To perform the opposite operation of that inferred from the operand type, explicit casts can be used. For example, if x is a variable of type int, the operation unchecked((int)((uint)x >> y)) performs a logical shift right of x.

7.10 Relational and Type-Testing Operators

The ==, !=, <, >, <=, >=, is, and as operators are called the relational and type-testing operators.

www.it-ebooks.info

relational-expression:
    shift-expression
    relational-expression `<` *shift-expression*
    relational-expression `>` *shift-expression*
    relational-expression `<=` *shift-expression*
    relational-expression `>=` *shift-expression*
    relational-expression *is* *type*
    relational-expression *as* *type*

equality-expression:
    relational-expression
    equality-expression `==` *relational-expression*
    equality-expression `!=` *relational-expression*

The `is` operator is described in §7.10.10 and the `as` operator is described in §7.10.11.

The `==`, `!=`, `<`, `>`, `<=`, and `>=` operators are ***comparison operators***.

If an operand of a comparison operator has the compile-time type `dynamic`, then the expression is dynamically bound (§7.2.2). In this case, the compile-time type of the expression is `dynamic`, and the resolution described below will take place at runtime using the runtime type of those operands that have the compile-time type `dynamic`.

For an operation of the form `x` *op* `y`, where *op* is a comparison operator, overload resolution (§7.3.4) is applied to select a specific operator implementation. The operands are converted to the parameter types of the selected operator, and the type of the result is the return type of the operator.

The predefined comparison operators are described in the following sections. All predefined comparison operators return a result of type `bool`, as described in the following table.

Operation	Result
`x == y`	`true` if `x` is equal to `y`, `false` otherwise
`x != y`	`true` if `x` is not equal to `y`, `false` otherwise
`x < y`	`true` if `x` is less than `y`, `false` otherwise
`x > y`	`true` if `x` is greater than `y`, `false` otherwise
`x <= y`	`true` if `x` is less than or equal to `y`, `false` otherwise
`x >= y`	`true` if `x` is greater than or equal to `y`, `false` otherwise

www.it-ebooks.info

7.10.1 Integer Comparison Operators

The predefined integer comparison operators are listed here:

```
bool operator ==(int x, int y);
bool operator ==(uint x, uint y);
bool operator ==(long x, long y);
bool operator ==(ulong x, ulong y);

bool operator !=(int x, int y);
bool operator !=(uint x, uint y);
bool operator !=(long x, long y);
bool operator !=(ulong x, ulong y);

bool operator <(int x, int y);
bool operator <(uint x, uint y);
bool operator <(long x, long y);
bool operator <(ulong x, ulong y);

bool operator >(int x, int y);
bool operator >(uint x, uint y);
bool operator >(long x, long y);
bool operator >(ulong x, ulong y);

bool operator <=(int x, int y);
bool operator <=(uint x, uint y);
bool operator <=(long x, long y);
bool operator <=(ulong x, ulong y);

bool operator >=(int x, int y);
bool operator >=(uint x, uint y);
bool operator >=(long x, long y);
bool operator >=(ulong x, ulong y);
```

Each of these operators compares the numeric values of the two integer operands and returns a `bool` value that indicates whether the particular relation is `true` or `false`.

7.10.2 Floating Point Comparison Operators

The predefined floating point comparison operators are listed here:

```
bool operator ==(float x, float y);
bool operator ==(double x, double y);

bool operator !=(float x, float y);
bool operator !=(double x, double y);

bool operator <(float x, float y);
bool operator <(double x, double y);

bool operator >(float x, float y);
bool operator >(double x, double y);

bool operator <=(float x, float y);
bool operator <=(double x, double y);

bool operator >=(float x, float y);
bool operator >=(double x, double y);
```


www.it-ebooks.info

These operators compare the operands according to the rules of the IEEE 754 standard:

- If either operand is NaN, the result is `false` for all operators except `!=`, for which the result is `true`. For any two operands, `x != y` always produces the same result as `!(x == y)`. However, when one or both operands are NaN, the `<`, `>`, `<=`, and `>=` operators *do not* produce the same results as the logical negation of the opposite operator. For example, if either of `x` and `y` is NaN, then `x < y` is `false`, but `!(x >= y)` is `true`.
- When neither operand is NaN, the operators compare the values of the two floating point operands with respect to the ordering

  ```
  −∞ < −max < ... < −min < −0.0 == +0.0 < +min < ... < +max < +∞
  ```

  where `min` and `max` are, respectively, the smallest and largest positive finite values that can be represented in the given floating point format. Notable effects of this ordering are as follows:

  - Negative and positive zeros are considered equal.
  - A negative infinity is considered less than all other values, but equal to another negative infinity.
  - A positive infinity is considered greater than all other values, but equal to another positive infinity.

PETER SESTOFT It also holds that positive and negative zero are equal by `Object.Equals`, so their hashcodes as computed by `Object.GetHashCode` should be equal as well. Unfortunately, this is not the case in all current implementations.

JOSEPH ALBAHARI Two NaN values, although unequal according to the `==` operator, are equal according to the `Equals` method:

```
double x = double.NaN;
Console.WriteLine(x == x);          // False
Console.WriteLine(x != x);          // True
Console.WriteLine(x.Equals(x));     // True
```

In general, a type's `Equals` method follows the principle that an object must equal itself. Without this assumption, lists and dictionaries could not operate, because there would be no means for testing element membership. The `==` and `!=` operators, however, are not obliged to follow this principle.

PETER SESTOFT Even NaNs that have different payloads (see the annotation on §7.8.1) are equal by `Object.Equals`.

www.it-ebooks.info

7.10.3 Decimal Comparison Operators

The predefined decimal comparison operators are listed here:

```
bool operator ==(decimal x, decimal y);
bool operator !=(decimal x, decimal y);
bool operator <(decimal x, decimal y);
bool operator >(decimal x, decimal y);
bool operator <=(decimal x, decimal y);
bool operator >=(decimal x, decimal y);
```

Each of these operators compares the numeric values of the two decimal operands and returns a `bool` value that indicates whether the particular relation is `true` or `false`. Each decimal comparison is equivalent to using the corresponding relational or equality operator of type `System.Decimal`.

7.10.4 Boolean Equality Operators

The predefined boolean equality operators are listed here:

```
bool operator ==(bool x, bool y);
bool operator !=(bool x, bool y);
```

The result of `==` is `true` if both `x` and `y` are `true` or if both `x` and `y` are `false`. Otherwise, the result is `false`.

The result of `!=` is `false` if both `x` and `y` are `true` or if both `x` and `y` are `false`. Otherwise, the result is `true`. When the operands are of type `bool`, the `!=` operator produces the same result as the `^` operator.

7.10.5 Enumeration Comparison Operators

Every enumeration type implicitly provides the following predefined comparison operators:

```
bool operator ==(E x, E y);
bool operator !=(E x, E y);
bool operator <(E x, E y);
bool operator >(E x, E y);
bool operator <=(E x, E y);
bool operator >=(E x, E y);
```

The result of evaluating `x` *op* `y`, where `x` and `y` are expressions of an enumeration type `E` with an underlying type `U`, and *op* is one of the comparison operators, is exactly the same as evaluating `((U)x)` *op* `((U)y)`. In other words, the enumeration type comparison operators simply compare the underlying integral values of the two operands.

www.it-ebooks.info

7.10.6 Reference Type Equality Operators

The predefined reference type equality operators are listed here:

```
bool operator ==(object x, object y);
bool operator !=(object x, object y);
```

The operators return the result of comparing the two references for equality or non-equality.

Since the predefined reference type equality operators accept operands of type `object`, they apply to all types that do not declare applicable `operator ==` and `operator !=` members. Conversely, any applicable user-defined equality operators effectively hide the predefined reference type equality operators.

The predefined reference type equality operators require one of the following:

- Both operands are a value of a type known to be a *reference-type* or the literal `null`. Furthermore, an explicit reference conversion (§6.2.4) exists from the type of either operand to the type of the other operand.
- One operand is a value of type `T`, where `T` is a *type-parameter* and the other operand is the literal `null`. Furthermore, `T` does not have the value type constraint.

Unless one of these conditions are true, a binding-time error occurs. Notable implications of these rules are as follows:

- It is a binding-time error to use the predefined reference type equality operators to compare two references that are known to be different at binding time. For example, if the binding-time types of the operands are two class types `A` and `B`, and if neither `A` nor `B` derives from the other, then it would be impossible for the two operands to reference the same object. Thus the operation is considered a binding-time error.
- The predefined reference type equality operators do not permit value type operands to be compared. Therefore, unless a struct type declares its own equality operators, it is not possible to compare values of that struct type.
- The predefined reference type equality operators never cause boxing operations to occur for their operands. It would be meaningless to perform such boxing operations, since references to the newly allocated boxed instances would necessarily differ from all other references.
- If an operand of a type parameter type `T` is compared to `null`, and the runtime type of `T` is a value type, the result of the comparison is `false`.

www.it-ebooks.info

The following example checks whether an argument of an unconstrained type parameter type is `null`.

```
class C<T>
{
    void F(T x) {
        if (x == null) throw new ArgumentNullException();
        ...
    }
}
```

The `x == null` construct is permitted even though `T` could represent a value type, and the result is simply defined to be `false` when `T` is a value type.

> **BILL WAGNER** If `C.F()` is called with `default(int?)`, it will throw an exception. A nullable type is considered equal to `null` if `HasValue` is `false`.

For an operation of the form `x == y` or `x != y`, if any applicable `operator ==` or `operator !=` exists, the operator overload resolution (§7.3.4) rules will select that operator instead of the predefined reference type equality operator. However, it is always possible to select the predefined reference type equality operator by explicitly casting one or both of the operands to type `object`. The example

```
using System;

class Test
{
    static void Main()
    {
        string s = "Test";
        string t = string.Copy(s);
        Console.WriteLine(s == t);
        Console.WriteLine((object)s == t);
        Console.WriteLine(s == (object)t);
        Console.WriteLine((object)s == (object)t);
    }
}
```

produces the output

```
True
False
False
False
```


www.it-ebooks.info

The `s` and `t` variables refer to two distinct `string` instances containing the same characters. The first comparison outputs `True` because the predefined string equality operator (§7.10.7) is selected when both operands are of type `string`. The remaining comparisons all output `False` because the predefined reference type equality operator is selected when one or both of the operands are of type `object`.

Note that the above technique is not meaningful for value types. The example

```
class Test
{
    static void Main()
    {
        int i = 123;
        int j = 123;
        System.Console.WriteLine((object)i == (object)j);
    }
}
```

outputs `False` because the casts create references to two separate instances of boxed `int` values.

7.10.7 String Equality Operators

The predefined string equality operators are listed here:

```
bool operator ==(string x, string y);
bool operator !=(string x, string y);
```

Two `string` values are considered equal when one of the following is true:

- Both values are `null`.
- Both values are non-null references to string instances that have identical lengths and identical characters in each character position.

The string equality operators compare string *values* rather than string *references*. When two separate string instances contain the exact same sequence of characters, the values of the strings are equal, but the references are different. As described in §7.10.6, the reference type equality operators can be used to compare string references instead of string values.

7.10.8 Delegate Equality Operators

Every delegate type implicitly provides the following predefined comparison operators:

```
bool operator ==(System.Delegate x, System.Delegate y);
bool operator !=(System.Delegate x, System.Delegate y);
```


www.it-ebooks.info

Two delegate instances are considered equal in the following circumstances:

- If either of the delegate instances is `null`, they are equal if and only if both are `null`.
- If the delegates have different runtime types, they are never equal.
- If both of the delegate instances have an invocation list (§15.1), those instances are equal if and only if their invocation lists are the same length, and each entry in one's invocation list is equal (as defined below) to the corresponding entry, in order, in the other's invocation list.

The following rules govern the equality of invocation list entries:

- If two invocation list entries both refer to the same static method, then the entries are equal.
- If two invocation list entries both refer to the same non-static method on the same target object (as defined by the reference equality operators), then the entries are equal.
- Invocation list entries produced from evaluation of semantically identical *anonymous-function-expression*s with the same (possibly empty) set of captured outer variable instances are permitted (but not required) to be equal.

7.10.9 Equality Operators and null

The `==` and `!=` operators permit one operand to be a value of a nullable type and the other to be the `null` literal, even if no predefined or user-defined operator (in unlifted or lifted form) exists for the operation.

For an operation of one of the forms

```
x == null    null == x    x != null    null != x
```

where `x` is an expression of a nullable type, if operator overload resolution (§7.2.4) fails to find an applicable operator, the result is instead computed from the `HasValue` property of `x`. Specifically, the first two forms are translated into `!x.HasValue`, and last two forms are translated into `x.HasValue`.

7.10.10 The is Operator

The `is` operator is used to dynamically check if the runtime type of an object is compatible with a given type. The result of the operation `E is T`, where `E` is an expression and `T` is a type, is a boolean value indicating whether `E` can successfully be converted to type `T` by a reference conversion, a boxing conversion, or an unboxing conversion. The operation is evaluated as follows, after type arguments have been substituted for all type parameters:

www.it-ebooks.info

- If E is an anonymous function, a compile-time error occurs.
- If E is a method group or the null literal, of if the type of E is a reference type or a nullable type and the value of E is null, the result is false.
- Otherwise, let D represent the dynamic type of E as follows:
  - If the type of E is a reference type, D is the runtime type of the instance reference by E.
  - If the type of E is a nullable type, D is the underlying type of that nullable type.
  - If the type of E is a non-nullable value type, D is the type of E.
- The result of the operation depends on D and T as follows:
  - If T is a reference type, the result is true if D and T are the same type, if D is a reference type and an implicit reference conversion from D to T exists, or if D is a value type and a boxing conversion from D to T exists.
  - If T is a nullable type, the result is true if D is the underlying type of T.
  - If T is a non-nullable value type, the result is true if D and T are the same type.
  - Otherwise, the result is false.

Note that user-defined conversions are not considered by the is operator.

7.10.11 The as Operator

The as operator is used to explicitly convert a value to a given reference type or nullable type. Unlike a cast expression (§7.7.6), the as operator never throws an exception. Instead, if the indicated conversion is not possible, the resulting value is null.

In an operation of the form E as T, E must be an expression and T must be a reference type, a type parameter known to be a reference type, or a nullable type. Furthermore, at least one of the following must be true, or otherwise a compile-time error occurs:

- An identity (§6.1.1), implicit nullable (§6.1.4), implicit reference (§6.1.6), boxing (§6.1.7), explicit nullable (§6.2.3), explicit reference (§6.2.4), or unboxing (§6.2.5) conversion exists from E to T.
- The type of E or T is an open type.
- E is the null literal.

If the compile-time type of E is not dynamic, the operation E as T produces the same result as

```
E is T ? (T)(E) : (T)null
```


www.it-ebooks.info

except that E is evaluated only once. The compiler can be expected to optimize E as T to perform at most one dynamic type check as opposed to the two dynamic type checks implied by the expansion above.

If the compile-time type of E is dynamic, unlike the cast operator, the as operator is not dynamically bound (§7.2.2). Therefore the expansion in this case is

```
E is T ? (T)(object)(E) : (T)null
```

Note that some conversions, such as user-defined conversions, are not possible with the as operator and should instead be performed using cast expressions.

In the example

```
class X
{
    public string F(object o) {
        return o as string;
            //Okay: string is a reference type
    }

    public T G<T>(object o) where T: Attribute {
        return o as T;
            // Okay: T has a class constraint
    }

    public U H<U>(object o) {
        return o as U;
            // Error: U is unconstrained
    }
}
```

the type parameter T of G is known to be a reference type, because it has the class constraint. The type parameter U of H is not known, however; hence the use of the as operator in H is disallowed.

CHRIS SELLS If you're using FxCop (and you should be!) out of the box, it doesn't like it when you do this:

```
if( x is Foo ) { ((Foo)x).DoFoo(); }
```

Instead, it prefers that you do this:

```
Foo foo = x as Foo;
if( foo != null ) { foo.DoFoo(); }
```

Why? Because is, as, and casting are all essentially the same underlying .NET operation, so it's better that you perform the operation once, check the result for null, and then do something with the non-null result than that you perform the operation twice.

www.it-ebooks.info

JOSEPH ALBAHARI Although Chris is technically right, this is a micro-optimization. Reference conversions are cheap.

CHRIS SELLS Although Joseph is absolutely right, I was talking more about keeping FxCop off my back than optimizing for performance.

JOSEPH ALBAHARI Some people prefer the `as` operator over the cast operator as a matter of style. An advantage of the `as` operator is that its use makes it clear that the conversion is not a numeric or user-defined conversion. The problem with universally favoring the `as` operator for reference conversions, however, is that it's not always desirable to have a failed conversion evaluate to `null`. To illustrate, consider the outcome of the following code if the object referenced by `s` is *not* a string:

```
int length1 = ((string) s).Length;   // Throws InvalidCastException
int length2 = (s as string).Length;  // Throws NullReferenceException
```

The first line throws a usefully populated `InvalidCastException`, whereas the second line throws an (ambiguous) `NullReferenceException`. (Was `s` `null` or the wrong type?)

JON SKEET The `as` operator is almost always used with a nullity test afterward—to the extent that it almost deserves its own statement type:

```
// Would be equivalent to the code in Chris's annotation
asif (Foo foo = x)
{
    foo.DoFoo();
}
```

Having a whole extra kind of statement would probably be overkill—but the clumsiness of the current pattern is irritating.

7.11 Logical Operators

The `&`, `^`, and `|` operators are called the logical operators.

and-expression:
    equality-expression
    and-expression `&` *equality-expression*

www.it-ebooks.info

exclusive-or-expression:
    and-expression
    exclusive-or-expression `^` *and-expression*

inclusive-or-expression:
    exclusive-or-expression
    inclusive-or-expression `|` *exclusive-or-expression*

If an operand of a logical operator has the compile-time type `dynamic`, then the expression is dynamically bound (§7.2.2). In this case, the compile-time type of the expression is `dynamic`, and the resolution described below will take place at runtime using the runtime type of those operands that have the compile-time type `dynamic`.

For an operation of the form `x` *op* `y`, where *op* is one of the logical operators, overload resolution (§7.3.4) is applied to select a specific operator implementation. The operands are converted to the parameter types of the selected operator, and the type of the result is the return type of the operator.

The predefined logical operators are described in the following sections.

7.11.1 Integer Logical Operators

The predefined integer logical operators are listed here:

```
int operator &(int x, int y);
uint operator &(uint x, uint y);
long operator &(long x, long y);
ulong operator &(ulong x, ulong y);

int operator |(int x, int y);
uint operator |(uint x, uint y);
long operator |(long x, long y);
ulong operator |(ulong x, ulong y);

int operator ^(int x, int y);
uint operator ^(uint x, uint y);
long operator ^(long x, long y);
ulong operator ^(ulong x, ulong y);
```

The `&` operator computes the bitwise logical `AND` of the two operands, the `|` operator computes the bitwise logical `OR` of the two operands, and the `^` operator computes the bitwise logical exclusive `OR` of the two operands. No overflows are possible from these operations.

7.11.2 Enumeration Logical Operators

Every enumeration type `E` implicitly provides the following predefined logical operators:

```
E operator &(E x, E y);
E operator |(E x, E y);
E operator ^(E x, E y);
```


www.it-ebooks.info

The result of evaluating `x` *op* `y`, where `x` and `y` are expressions of an enumeration type `E` with an underlying type `U`, and *op* is one of the logical operators, is exactly the same as evaluating `(E)((U)x` *op* `(U)y)`. In other words, the enumeration type logical operators simply perform the logical operation on the underlying type of the two operands.

7.11.3 Boolean Logical Operators

The predefined boolean logical operators are listed here:

```
bool operator &(bool x, bool y);
bool operator |(bool x, bool y);
bool operator ^(bool x, bool y);
```

The result of `x & y` is `true` if both `x` and `y` are `true`. Otherwise, the result is `false`.

The result of `x | y` is `true` if either `x` or `y` is `true`. Otherwise, the result is `false`.

The result of `x ^ y` is `true` if `x` is `true` and `y` is `false`, or if `x` is `false` and `y` is `true`. Otherwise, the result is `false`. When the operands are of type `bool`, the `^` operator computes the same result as the `!=` operator.

7.11.4 Nullable Boolean Logical Operators

The nullable boolean type `bool?` can represent three values: `true`, `false`, and `null`. It is conceptually similar to the three-valued type used for boolean expressions in SQL. To ensure that the results produced by the `&` and `|` operators for `bool?` operands are consistent with SQL's three-valued logic, the following predefined operators are provided:

```
bool? operator &(bool? x, bool? y);
bool? operator |(bool? x, bool? y);
```

The following table lists the results produced by these operators for all combinations of the values `true`, `false`, and `null`.

x	y	x & y	x | y
true	true	true	true
true	false	false	true
true	null	null	true
false	true	false	true
false	false	false	false
false	null	false	null
null	true	null	true
null	false	false	null
null	null	null	null

www.it-ebooks.info

7.12 Conditional Logical Operators

The `&&` and `||` operators are called the conditional logical operators. They are also called the "short-circuiting" logical operators.

conditional-and-expression:
inclusive-or-expression
conditional-and-expression `&&` *inclusive-or-expression*

conditional-or-expression:
conditional-and-expression
conditional-or-expression `||` *conditional-and-expression*

The `&&` and `||` operators are conditional versions of the `&` and `|` operators:

- The operation `x && y` corresponds to the operation `x & y`, except that `y` is evaluated only if `x` is not `false`.
- The operation `x || y` corresponds to the operation `x | y`, except that `y` is evaluated only if `x` is not `true`.

If an operand of a conditional logical operator has the compile-time type `dynamic`, then the expression is dynamically bound (§7.2.2). In this case, the compile-time type of the expression is `dynamic`, and the resolution described below will take place at runtime using the runtime type of those operands that have the compile-time type `dynamic`.

An operation of the form `x && y` or `x || y` is processed by applying overload resolution (§7.3.4) as if the operation was written `x & y` or `x | y`. Then,

- If overload resolution fails to find a single best operator, or if overload resolution selects one of the predefined integer logical operators, a binding-time error occurs.
- Otherwise, if the selected operator is one of the predefined boolean logical operators (§7.11.3) or nullable boolean logical operators (§7.11.4), the operation is processed as described in §7.12.1.
- Otherwise, the selected operator is a user-defined operator, and the operation is processed as described in §7.12.2.

It is not possible to directly overload the conditional logical operators. However, because the conditional logical operators are evaluated in terms of the regular logical operators, overloads of the regular logical operators are, with certain restrictions, also considered overloads of the conditional logical operators. This situation is described further in §7.12.2.

www.it-ebooks.info

7.12.1 Boolean Conditional Logical Operators

When the operands of `&&` or `||` are of type `bool`, or when the operands are of types that do not define an applicable `operator &` or `operator |` but do define implicit conversions to `bool`, the operation is processed as follows:

- The operation `x && y` is evaluated as `x ? y : false`. In other words, `x` is first evaluated and converted to type `bool`. Then, if `x` is `true`, `y` is evaluated and converted to type `bool`, and this becomes the result of the operation. Otherwise, the result of the operation is `false`.
- The operation `x || y` is evaluated as `x ? true : y`. In other words, `x` is first evaluated and converted to type `bool`. Then, if `x` is `true`, the result of the operation is `true`. Otherwise, `y` is evaluated and converted to type `bool`, and this becomes the result of the operation.

7.12.2 User-Defined Conditional Logical Operators

When the operands of `&&` or `||` are of types that declare an applicable user-defined `operator &` or `operator |`, both of the following must be true, where `T` is the type in which the selected operator is declared:

- The return type and the type of each parameter of the selected operator must be `T`. In other words, the operator must compute the logical `AND` or the logical `OR` of two operands of type `T`, and must return a result of type `T`.
- `T` must contain declarations of `operator true` and `operator false`.

A binding-time error occurs if either of these requirements is not satisfied. Otherwise, the `&&` or `||` operation is evaluated by combining the user-defined `operator true` or `operator false` with the selected user-defined operator:

- The operation `x && y` is evaluated as `T.false(x) ? x : T.&(x, y)`, where `T.false(x)` is an invocation of the `operator false` declared in `T`, and `T.&(x, y)` is an invocation of the selected `operator &`. In other words, `x` is first evaluated and `operator false` is invoked on the result to determine if `x` is definitely false. Then, if `x` is definitely false, the result of the operation is the value previously computed for `x`. Otherwise, `y` is evaluated, and the selected `operator &` is invoked on the value previously computed for `x` and the value computed for `y` to produce the result of the operation.
- The operation `x || y` is evaluated as `T.true(x) ? x : T.|(x, y)`, where `T.true(x)` is an invocation of the `operator true` declared in `T`, and `T.|(x, y)` is an invocation of the selected `operator |`. In other words, `x` is first evaluated and `operator true` is invoked on the result to determine if `x` is definitely true. Then, if `x` is definitely true, the result of the operation is the value previously computed for `x`. Otherwise, `y` is evaluated, and the selected `operator |` is invoked on the value previously computed for `x` and the value computed for `y` to produce the result of the operation.

www.it-ebooks.info

In either of these operations, the expression given by `x` is evaluated only once, and the expression given by `y` is either not evaluated or evaluated exactly once.

For an example of a type that implements `operator true` and `operator false`, see §11.4.2.

7.13 The Null Coalescing Operator

The `??` operator is called the null coalescing operator.

null-coalescing-expression:
    conditional-or-expression
    conditional-or-expression `??` *null-coalescing-expression*

A null coalescing expression of the form `a ?? b` requires `a` to be of a nullable type or reference type. If `a` is non-null, the result of `a ?? b` is `a`; otherwise, the result is `b`. The operation evaluates `b` only if `a` is null.

The null coalescing operator is right-associative, meaning that operations are grouped from right to left. For example, an expression of the form `a ?? b ?? c` is evaluated as `a ?? (b ?? c)`. In general terms, an expression of the form E_1 `??` E_2 `??` ... `??` E_N returns the first of the operands that is non-null, or returns `null` if all operands are null.

The type of the expression `a ?? b` depends on which implicit conversions are available on the operands. In order of preference, the type of `a ?? b` is A_θ, `A`, or `B`, where `A` is the type of `a` (provided that `a` has a type), `B` is the type of `b` (provided that `b` has a type), and A_θ is the underlying type of `A` if `A` is a nullable type, or `A` otherwise. Specifically, `a ?? b` is processed as follows:

- If `A` exists and is not a nullable type or a reference type, a compile-time error occurs.
- If `b` is a dynamic expression, the result type is `dynamic`. At runtime, `a` is first evaluated. If `a` is not null, `a` is converted to a `dynamic` type, and this becomes the result. Otherwise, `b` is evaluated, and the outcome becomes the result.
- Otherwise, if `A` exists and is a nullable type and an implicit conversion exists from `b` to A_θ, the result type is A_θ. At runtime, `a` is first evaluated. If `a` is not null, `a` is unwrapped to type A_θ, and it becomes the result. Otherwise, `b` is evaluated and converted to type A_θ, and it becomes the result.
- Otherwise, if `A` exists and an implicit conversion exists from `b` to `A`, the result type is `A`. At runtime, `a` is first evaluated. If `a` is not null, `a` becomes the result. Otherwise, `b` is evaluated and converted to type `A`, and it becomes the result.

www.it-ebooks.info

- Otherwise, if `b` has a type `B` and an implicit conversion exists from `a` to `B`, the result type is `B`. At runtime, `a` is first evaluated. If `a` is not null, `a` is unwrapped to type A_0 (if `A` exists and is nullable) and converted to type `B`, and it becomes the result. Otherwise, `b` is evaluated and becomes the result.
- Otherwise, `a` and `b` are incompatible, and a compile-time error occurs.

ERIC LIPPERT These conversion rules considerably complicate the transformation of a null coalescing operator into an expression tree. In some cases, the compiler must emit an additional expression tree lambda specifically to handle the conversion logic.

CHRIS SELLS The `??` operator is useful for setting default values for reference types or nullable value types. For example:

```
Foo f1 = ...;
Foo f2 = f1 ?? new Foo(...);
int? i1 = ...;
int i2 = i1 ?? 452;
```

7.14 Conditional Operator

The `?:` operator is called the conditional operator or, sometimes, the ternary operator.

conditional-expression:
null-coalescing-expression
null-coalescing-expression `?` *expression* `:` *expression*

A conditional expression of the form `b ? x : y` first evaluates the condition `b`. Then, if `b` is `true`, `x` is evaluated and becomes the result of the operation. Otherwise, `y` is evaluated and becomes the result of the operation. A conditional expression never evaluates both `x` and `y`.

The conditional operator is right-associative, meaning that operations are grouped from right to left. For example, an expression of the form `a ? b : c ? d : e` is evaluated as `a ? b : (c ? d : e)`.

www.it-ebooks.info

JON SKEET I find myself using multiple conditional operators occasionally. It looks odd at first, but can be very readable when laid out appropriately. In some ways, this is the closest C# gets to the pattern matching of languages such as F#.

```
return firstCondition ? firstValue :
       secondCondition ? secondValue :
       thirdCondition ? thirdValue :
       fallbackValue;
```

The first operand of the `?:` operator must be an expression that can be implicitly converted to `bool`, or an expression of a type that implements `operator true`. If neither of these requirements is satisfied, a compile-time error occurs.

PETER SESTOFT In particular, because there is no implicit conversion from the nullable type `bool?` to `bool`, the first operand of the conditional operator cannot have type `bool?`.

The second and third operands, `x` and `y`, of the `?:` operator control the type of the conditional expression.

- If `x` has type `X` and `y` has type `Y`, then
  - If an implicit conversion (§6.1) exists from `X` to `Y`, but not from `Y` to `X`, then `Y` is the type of the conditional expression.
  - If an implicit conversion (§6.1) exists from `Y` to `X`, but not from `X` to `Y`, then `X` is the type of the conditional expression.
  - Otherwise, no expression type can be determined, and a compile-time error occurs.
- If only one of `x` and `y` has a type, and both `x` and `y` are implicitly convertible to that type, then that is the type of the conditional expression.
- Otherwise, no expression type can be determined, and a compile-time error occurs.

ERIC LIPPERT The Microsoft C# compiler actually implements a slightly different algorithm: It checks for conversions from the *expressions* to the types, not from *types* to types. In most cases, the difference does not matter and it would break existing code to change it now.

www.it-ebooks.info

PETER SESTOFT Despite the point made in Eric's annotation, there's a small incompatibility between Microsoft's C# 4.0 compiler and previous versions: The compiler cannot infer the type of the conditional expression in the right-hand side below, complaining that "there is no implicit conversion between '<null>' and '<null>'":

```
int? x = args.Length>0 ? null : null;
```

The C# 2.0 compiler would accept this expression. The C# 4.0 compiler's message hints at the possible existence of a "null type" inside the compiler, or at least in the minds of those who wrote the error message. Given the extremely contrived piece of code, the incompatibility is nothing to lose sleep over.

VLADIMIR RESHETNIKOV A notion of "null type" existed in earlier versions of the C# specification, but was eliminated in C# 3.0, which resulted in a slightly different behavior in corner cases. Currently, `null` literal is an expression that has no type.

The runtime processing of a conditional expression of the form `b ? x : y` consists of the following steps:

- First, `b` is evaluated, and the `bool` value of `b` is determined:
  - If an implicit conversion from the type of `b` to `bool` exists, then this implicit conversion is performed to produce a `bool` value.
  - Otherwise, the `operator true` defined by the type of `b` is invoked to produce a `bool` value.
- If the `bool` value produced by the step above is `true`, then `x` is evaluated and converted to the type of the conditional expression, and this becomes the result of the conditional expression.
- Otherwise, `y` is evaluated and converted to the type of the conditional expression, and this becomes the result of the conditional expression.

BILL WAGNER To me, both the null coalescing operator and the conditional operator are like strong spices: When used sparingly, they are great enhancers. Chris points out usages where the null coalescing operator creates clearer code than the equivalent `if-then-else` statements. Like strong spices, overuse is painful and ruins what could otherwise be very pleasant.

www.it-ebooks.info

7.15 Anonymous Function Expressions

An ***anonymous function*** is an expression that represents an "in-line" method definition. An anonymous function does not have a value or type in and of itself, but is convertible to a compatible delegate or expression tree type. The evaluation of an anonymous function conversion depends on the target type of the conversion: If it is a delegate type, the conversion evaluates to a delegate value referencing the method that the anonymous function defines. If it is an expression tree type, the conversion evaluates to an expression tree that represents the structure of the method as an object structure.

For historical reasons, there are two syntactic flavors of anonymous functions—namely, *lambda-expression*s and *anonymous-method-expression*s. For almost all purposes, *lambda-expression*s are more concise and expressive than *anonymous-method-expression*s, which remain in the language to ensure backward compatibility.

lambda-expression:
    anonymous-function-signature `=>` *anonymous-function-body*

anonymous-method-expression:
    `delegate` *explicit-anonymous-function-signature*$_{opt}$ *block*

anonymous-function-signature:
    explicit-anonymous-function-signature
    implicit-anonymous-function-signature

explicit-anonymous-function-signature:
    `(` *explicit-anonymous-function-parameter-list*$_{opt}$ `)`

explicit-anonymous-function-parameter-list:
    explicit-anonymous-function-parameter
    explicit-anonymous-function-parameter-list `,` *explicit-anonymous-function-parameter*

explicit-anonymous-function-parameter:
    anonymous-function-parameter-modifier$_{opt}$ *type* *identifier*

anonymous-function-parameter-modifier:
    `ref`
    `out`

implicit-anonymous-function-signature:
    `(` *implicit-anonymous-function-parameter-list*$_{opt}$ `)`
    implicit-anonymous-function-parameter

www.it-ebooks.info

implicit-anonymous-function-parameter-list:
implicit-anonymous-function-parameter
implicit-anonymous-function-parameter-list , *implicit-anonymous-function-parameter*

implicit-anonymous-function-parameter:
identifier

anonymous-function-body:
expression
block

The => operator has the same precedence as assignment (=) and is right-associative.

The parameters of an anonymous function in the form of a *lambda-expression* can be explicitly or implicitly typed. In an explicitly typed parameter list, the type of each parameter is explicitly stated. In an implicitly typed parameter list, the types of the parameters are inferred from the context in which the anonymous function occurs—specifically, when the anonymous function is converted to a compatible delegate type or expression tree type, that type provides the parameter types (§6.5).

BILL WAGNER In general, your anonymous functions will be more resilient if you rely on implicit typing.

In an anonymous function with a single, implicitly typed parameter, the parentheses may be omitted from the parameter list. In other words, an anonymous function of the form

(*param*) => *expr*

can be abbreviated as

param => *expr*

The parameter list of an anonymous function in the form of an *anonymous-method-expression* is optional. If given, the parameters must be explicitly typed. If not, the anonymous function is convertible to a delegate with any parameter list not containing out parameters.

Some examples of anonymous functions follow below:

```
x => x + 1                          // Implicitly typed, expression body
x => { return x + 1; }              // Implicitly typed, statement body
(int x) => x + 1                    // Explicitly typed, expression body
(int x) => { return x + 1; }        // Explicitly typed, statement body
(x, y) => x * y                     // Multiple parameters
() => Console.WriteLine()           // No parameters
delegate (int x) { return x + 1; }  // Anonymous method expression
delegate { return 1 + 1; }          // Parameter list omitted
```


www.it-ebooks.info

The behavior of *lambda-expression*s and *anonymous-method-expression*s is the same except for the following points:

- The *anonymous-method-expression*s permit the parameter list to be omitted entirely, yielding convertibility to delegate types of any list of value parameters.
- The *lambda-expression*s permit parameter types to be omitted and inferred, whereas the *anonymous-method-expression*s require parameter types to be explicitly stated.
- The body of a *lambda-expression* can be an expression or a statement block, whereas the body of an *anonymous-method-expression* must be a statement block.
- Since only *lambda-expression*s can have an *expression* body, no *anonymous-method-expression* can be successfully converted to an expression tree type (§4.6).

BILL WAGNER This point is important for building queries that rely on expression trees, such as those in Linq2SQL and Entity Framework.

7.15.1 Anonymous Function Signatures

The optional *anonymous-function-signature* of an anonymous function defines the names and optionally the types of the formal parameters for the anonymous function. The scope of the parameters of the anonymous function is the *anonymous-function-body* (§3.7). Together with the parameter list (if given), the *anonymous-method-body* constitutes a declaration space (§3.3). It is thus a compile-time error for the name of a parameter of the anonymous function to match the name of a local variable, local constant, or parameter whose scope includes the *anonymous-method-expression* or *lambda-expression*.

If an anonymous function has an *explicit-anonymous-function-signature*, then the set of compatible delegate types and expression tree types is restricted to those that have the same parameter types and modifiers in the same order. In contrast to method group conversions (§6.6), contravariance of anonymous function parameter types is not supported. If an anonymous function does not have an *anonymous-function-signature*, then the set of compatible delegate types and expression tree types is restricted to those that have no `out` parameters.

Note that an *anonymous-function-signature* cannot include attributes or a parameter array. Nevertheless, an *anonymous-function-signature* may be compatible with a delegate type whose parameter list contains a parameter array.

Note also that conversion to an expression tree type, even if compatible, may still fail at compile time (§4.6).

www.it-ebooks.info

ERIC LIPPERT This is a subtle point. If you have two overloads—say, `void M(Expression<Func<Giraffe>> f)` and `void M(Func<Animal> f)`—and a call `M(()=>myGiraffes[++i])`, then the expression tree overload is chosen as the better overload. In this situation, a compile-time error occurs because the increment operator is illegal inside an expression tree.

7.15.2 Anonymous Function Bodies

The body (*expression* or *block*) of an anonymous function is subject to the following rules:

- If the anonymous function includes a signature, the parameters specified in the signature are available in the body. If the anonymous function has no signature, it can be converted to a delegate type or expression type having parameters (§6.5), but the parameters cannot be accessed in the body.
- Except for `ref` or `out` parameters specified in the signature (if any) of the nearest enclosing anonymous function, it is a compile-time error for the body to access a `ref` or `out` parameter.
- When the type of `this` is a struct type, it is a compile-time error for the body to access `this`. This is true whether the access is explicit (as in `this.x`) or implicit (as in `x` where `x` is an instance member of the struct). This rule simply prohibits such access and does not affect whether member lookup results in a member of the struct.
- The body has access to the outer variables (§7.15.5) of the anonymous function. Access to an outer variable will reference the instance of the variable that is active at the time the *lambda-expression* or *anonymous-method-expression* is evaluated (§7.15.6).
- It is a compile-time error for the body to contain a `goto` statement, `break` statement, or `continue` statement whose target is outside the body or within the body of a contained anonymous function.
- A `return` statement in the body returns control from an invocation of the nearest enclosing anonymous function, not from the enclosing function member. An expression specified in a `return` statement must be implicitly convertible to the return type of the delegate type or expression tree type to which the nearest enclosing *lambda-expression* or *anonymous-method-expression* is converted (§6.5).

It is explicitly unspecified whether there is any way to execute the block of an anonymous function other than through evaluation and invocation of the *lambda-expression* or *anonymous-method-expression*. In particular, the compiler may choose to implement an anonymous function by synthesizing one or more named methods or types. The names of any such synthesized elements must be of a form reserved for compiler use.

www.it-ebooks.info

7.15.3 Overload Resolution

Anonymous functions in an argument list participate in type inference and overload resolution. Please refer to §7.4.2.3 for the exact rules.

The following example illustrates the effect of anonymous functions on overload resolution.

```
class ItemList<T>: List<T>
{
    public int Sum(Func<T,int> selector) {
        int sum = 0;
        foreach (T item in this) sum += selector(item);
        return sum;
    }

    public double Sum(Func<T,double> selector) {
        double sum = 0;
        foreach (T item in this) sum += selector(item);
        return sum;
    }
}
```

The `ItemList<T>` class has two `Sum` methods. Each takes a `selector` argument, which extracts the value to sum over from a list item. The extracted value can be either an `int` or a `double`, and the resulting sum is likewise either an `int` or a `double`.

The `Sum` methods could, for example, be used to compute sums from a list of detail lines in an order.

```
class Detail
{
    public int UnitCount;
    public double UnitPrice;
    ...
}

void ComputeSums() {
    ItemList<Detail> orderDetails = GetOrderDetails(...);
    int totalUnits = orderDetails.Sum(d => d.UnitCount);
    double orderTotal = orderDetails.Sum(d => d.UnitPrice * d.UnitCount);
    ...
}
```

In the first invocation of `orderDetails.Sum`, both `Sum` methods are applicable because the anonymous function `d => d.UnitCount` is compatible with both `Func<Detail,int>` and `Func<Detail,double>`. However, overload resolution picks the first `Sum` method because the conversion to `Func<Detail,int>` is better than the conversion to `Func<Detail,double>`.

In the second invocation of `orderDetails.Sum`, only the second `Sum` method is applicable because the anonymous function `d => d.UnitPrice * d.UnitCount` produces a value of type `double`. Thus overload resolution picks the second `Sum` method for that invocation.

www.it-ebooks.info

7.15.4 Anonymous Functions and Dynamic Binding

An anonymous function cannot be a receiver, argument, or operand of a dynamically bound operation.

7.15.5 Outer Variables

Any local variable, value parameter, or parameter array whose scope includes the *lambda-expression* or *anonymous-method-expression* is called an ***outer variable*** of the anonymous function. In an instance function member of a class, the `this` value is considered a value parameter and is an outer variable of any anonymous function contained within the function member.

> **BILL WAGNER** This is the formal definition of how closures are implemented in C#. It's a great addition.

7.15.5.1 *Captured Outer Variables*

When an outer variable is referenced by an anonymous function, the outer variable is said to have been ***captured*** by the anonymous function. Ordinarily, the lifetime of a local variable is limited to execution of the block or statement with which it is associated (§5.1.7). However, the lifetime of a captured outer variable is extended at least until the delegate or expression tree created from the anonymous function becomes eligible for garbage collection.

> **BILL WAGNER** In the next example, notice that x has a longer life than you would expect, because it is captured by the anonymous method result. If x were an expensive resource, that behavior should be avoided by limiting the lifetime of the anonymous method.

In the example

```
using System;

delegate int D();

class Test
{
    static D F() {
        int x = 0;
        D result = () => ++x;
        return result;
    }
```


www.it-ebooks.info

```
        static void Main() {
            D d = F();
            Console.WriteLine(d());
            Console.WriteLine(d());
            Console.WriteLine(d());
        }
    }
```

the local variable x is captured by the anonymous function, and the lifetime of x is extended at least until the delegate returned from F becomes eligible for garbage collection (which doesn't happen until the very end of the program). Since each invocation of the anonymous function operates on the same instance of x, the output of the example is

```
1
2
3
```

When a local variable or a value parameter is captured by an anonymous function, the local variable or parameter is no longer considered to be a fixed variable (§18.3), but instead is considered to be a moveable variable. Thus any `unsafe` code that takes the address of a captured outer variable must first use the `fixed` statement to fix the variable.

Note that unlike an uncaptured variable, a captured local variable can be simultaneously exposed to multiple threads of execution.

7.15.5.2 *Instantiation of Local Variables*

A local variable is considered to be ***instantiated*** when execution enters the scope of the variable. For example, when the following method is invoked, the local variable x is instantiated and initialized three times—once for each iteration of the loop:

```
static void F() {
    for (int i = 0; i < 3; i++) {
        int x = i * 2 + 1;
        ...
    }
}
```

However, moving the declaration of x outside the loop results in a single instantiation of x:

```
static void F() {
    int x;
    for (int i = 0; i < 3; i++) {
        x = i * 2 + 1;
        ...
    }
}
```


www.it-ebooks.info

When not captured, there is no way to observe exactly how often a local variable is instantiated: Because the lifetimes of the instantiations are disjoint, it is possible for each instantiation to simply use the same storage location. However, when an anonymous function captures a local variable, the effects of instantiation become apparent.

The example

```
using System;

delegate void D();

class Test
{
    static D[] F() {
        D[] result = new D[3];
        for (int i = 0; i < 3; i++) {
            int x = i * 2 + 1;
            result[i] = () => { Console.WriteLine(x); };
        }
        return result;
    }

    static void Main() {
        foreach (D d in F()) d();
    }
}
```

produces the following output:

```
1
3
5
```

However, when the declaration of x is moved outside the loop

```
static D[] F() {
    D[] result = new D[3];
    int x;
    for (int i = 0; i < 3; i++) {
        x = i * 2 + 1;
        result[i] = () => { Console.WriteLine(x); };
    }
    return result;
}
```

the output is

```
5
5
5
```

If a `for` loop declares an iteration variable, that variable itself is considered to be declared outside of the loop. Thus, if the example is changed to capture the iteration variable itself,

www.it-ebooks.info

```
static D[] F() {
    D[] result = new D[3];
    for (int i = 0; i < 3; i++) {
        result[i] = () => { Console.WriteLine(i); };
    }
    return result;
}
```

only one instance of the iteration variable is captured, which produces the following output:

```
3
3
3
```

> **ERIC LIPPERT** This behavior— that anonymous functions capture loop variables, not the current value of that variable—is the cause of the single most common incorrect "I think I found a bug in the compiler" report we get. For a future version of this specification, we are considering moving the formal definition of the `foreach` loop variable to the inside of the loop, so that an anonymous function inside the loop would capture a new variable every time through the loop. That would technically be a breaking change, but it would bring the language in line with what most people think its behavior ought to be. It's hard to think of a situation in which you *want* to capture the loop variable as a variable, rather than as a value.

It is possible for anonymous function delegates to share some captured variables, yet have separate instances of others. For example, if `F` is changed to

```
D[] result = new D[3];
int x = 0;
for (int i = 0; i < 3; i++) {
    int y = 0;
    result[i] = () => { Console.WriteLine("{0} {1}", ++x, ++y); };
}
return result;
}
```

the three delegates capture the same instance of `x` but separate instances of `y`, and the output is

```
1 1
2 1
3 1
```

Separate anonymous functions can capture the same instance of an outer variable. In the example

```
using System;
```


www.it-ebooks.info

```
delegate void Setter(int value);

delegate int Getter();

class Test
{
    static void Main() {
        int x = 0;
        Setter s = (int value) => { x = value; };
        Getter g = () => { return x; };
        s(5);
        Console.WriteLine(g());
        s(10);
        Console.WriteLine(g());
    }
}
```

the two anonymous functions capture the same instance of the local variable `x`, and they can "communicate" through that variable. The output of the example is

```
5
10
```

7.15.6 Evaluation of Anonymous Function Expressions

An anonymous function `F` must always be converted to a delegate type `D` or an expression tree type `E`, either directly or through the execution of a delegate creation expression `new D(F)`. This conversion determines the result of the anonymous function, as described in §6.5.

7.16 Query Expressions

Query expressions provide a language-integrated syntax for queries that is similar to relational and hierarchical query languages such as SQL and XQuery.

query-expression:
    from-clause *query-body*

from-clause:
    `from` *type*$_{opt}$ *identifier* `in` *expression*

query-body:
    query-body-clauses$_{opt}$ *select-or-group-clause* *query-continuation*$_{opt}$

query-body-clauses:
    query-body-clause
    query-body-clauses *query-body-clause*

www.it-ebooks.info

query-body-clause:
    from-clause
    let-clause
    where-clause
    join-clause
    join-into-clause
    orderby-clause

let-clause:
    `let` *identifier* `=` *expression*

where-clause:
    `where` *boolean-expression*

join-clause:
    `join` *type*$_{opt}$ *identifier* `in` *expression* `on` *expression* `equals` *expression*

join-into-clause:
    `join` *type*$_{opt}$ *identifier* `in` *expression* `on` *expression* `equals` *expression* `into` *identifier*

orderby-clause:
    `orderby` *orderings*

orderings:
    ordering
    orderings `,` *ordering*

ordering:
    expression *ordering-direction*$_{opt}$

ordering-direction:
    `ascending`
    `descending`

select-or-group-clause:
    select-clause
    group-clause

select-clause:
    `select` *expression*

www.it-ebooks.info

group-clause:
`group` *expression* `by` *expression*

query-continuation:
`into` *identifier* *query-body*

A query expression begins with a `from` clause and ends with either a `select` or `group` clause. The initial `from` clause can be followed by zero or more `from`, `let`, `where`, `join`, or `orderby` clauses. Each `from` clause is a generator introducing a ***range variable*** that ranges over the elements of a ***sequence***. Each `let` clause introduces a range variable representing a value computed by means of previous range variables. Each `where` clause is a filter that excludes items from the result. Each `join` clause compares specified keys of the source sequence with keys of another sequence, yielding matching pairs. Each `orderby` clause reorders items according to specified criteria. The final `select` or `group` clause specifies the shape of the result in terms of the range variables. Finally, an `into` clause can be used to "splice" queries by treating the results of one query as a generator in a subsequent query.

ERIC LIPPERT Why is the query syntax in C# "`from...where...select`," rather than the order more familiar to SQL developers: "`select...from...where`"? Although the SQL order has the benefits of being familiar to SQL developers and natural to English speakers, it causes many problems for the language and IDE designers that are not present in the C# order.

First, IntelliSense is difficult to use with the SQL order because of the scoping rules. Imagine you're designing an IDE for the "`select` first" syntax. The user has just typed "`select`": Now what? You don't know which data source the user is projecting from, so you cannot produce a sensible list of helpful options. Nor can you provide any help for "`where`." In the C# system, the "`from`" comes before the "`select`"; thus, by the time the user types "`where`" or "`select`," the IDE knows type information about the data source.

Second, the C# order is the order in which operations are actually performed by the code: First a collection is identified, then the filter is run over the collection, and then the results of the filter are projected. This order helps give a sense of how data flows through the system.

CHRIS SELLS Now that I've seen both orders "`select-from`" in SQL and "`from-select`" in LINQ, LINQ makes much more sense to me and I'm frustrated that SQL doesn't allow the "`from-select`" style.

www.it-ebooks.info

7.16.1 Ambiguities in Query Expressions

Query expressions contain a number of "contextual keywords"—that is, identifiers that have special meaning in a given context. Specifically these are `from`, `where`, `join`, `on`, `equals`, `into`, `let`, `orderby`, `ascending`, `descending`, `select`, `group`, and `by`. To avoid ambiguities in query expressions caused by mixed use of these identifiers as keywords or simple names, these identifiers are considered keywords when occurring anywhere within a query expression.

For this purpose, a query expression is any expression that starts with "`from` *identifier*" followed by any token except "`;`", "`=`", or "`,`".

To use these words as identifiers within a query expression, they can be prefixed with "`@` (§2.4.2).

7.16.2 Query Expression Translation

The C# language does not specify the execution semantics of query expressions. Rather, query expressions are translated into invocations of methods that adhere to the *query expression pattern* (§7.16.3). Specifically, query expressions are translated into invocations of methods named `Where`, `Select`, `SelectMany`, `Join`, `GroupJoin`, `OrderBy`, `OrderByDescending`, `ThenBy`, `ThenByDescending`, `GroupBy`, and `Cast`. These methods are expected to have particular signatures and result types, as described in §7.16.3. These methods can be instance methods of the object being queried or extension methods that are external to the object, and they implement the actual execution of the query.

The translation from query expressions to method invocations is a syntactic mapping that occurs before any type binding or overload resolution has been performed. The translation is guaranteed to be syntactically correct, but is not guaranteed to produce semantically correct C# code. Following translation of query expressions, the resulting method invocations are processed as regular method invocations, and this may in turn uncover errors—for example, if the methods do not exist, if arguments have wrong types, or if the methods are generic and type inference fails.

> **BILL WAGNER** This entire section is a great way to understand how query expressions are translated into method calls and possibly extension method calls.

> **JON SKEET** The fact that query expressions could be introduced into C# with such a small addition to the specification is a source of wonder to me. Whereas some features, such as generics, affected nearly every area of the language, query expressions are amazingly self-contained—especially considering the expressive power they provide.

www.it-ebooks.info

A query expression is processed by repeatedly applying the following translations until no further reductions are possible. The translations are listed in order of application: Each section assumes that the translations in the preceding sections have been performed exhaustively; once exhausted, a section will not later be revisited in the processing of the same query expression.

Assignment to range variables is not allowed in query expressions. However, a C# implementation is permitted to not always enforce this restriction, since this may sometimes not be possible with the syntactic translation scheme presented here.

Certain translations inject range variables with *transparent identifiers* denoted by *. The special properties of transparent identifiers are discussed further in §7.16.2.7.

CHRIS SELLS As much as I like the C# 3.0 query syntax, sometimes it's difficult to keep the translations in my head. Don't feel bad if you occasionally feel the need to write out your queries using the method call syntax. Also, any query methods that you implement yourself will not have language constructs, so sometimes you won't have any choice except to use the method call syntax.

Further, it's completely okay if your peers are writing queries like this:

```
var duluthians = from c in Customers
         where c.City == "Duluth" select c;
```

and you write your queries like this:

```
 var duluthians = Customers.Where(c => c.City == "Duluth");
```

The second syntax requires no mental translation and works for all extension methods, not just the ones that have been translated into keywords. For example:

```
var tenDuluthites = Customers.Where(c => c.City == "Duluth").Take(10);
```

versus

```
var tenDuluthites = (from c in Customers where c.City ==
                                "Duluth" select c).Take(10);
```

I'm a simple man, so I prefer the syntax style that always works over what the cool kids are doing these days.

www.it-ebooks.info

JON SKEET While additional query methods on `IEnumerable<T>` are unlikely to be supported by query expressions, one of the beautiful aspects of their definition is the neutrality involved: There's nothing in the specification to dictate what the types should be. This has allowed other frameworks (such as Reactive Extensions and Parallel Extensions) to write query methods against new types and still take advantage of query expression syntax.

7.16.2.1 *select and GroupBy Clauses with Continuations*

A query expression with a continuation

```
from ... into x ...
```

is translated into

```
from x in ( from ... ) ...
```

The translations in the following sections assume that queries have no `into` continuations.

The example

```
from c in customers
group c by c.Country into g
select new { Country = g.Key, CustCount = g.Count() }
```

is translated into

```
from g in
    from c in customers
    group c by c.Country
select new { Country = g.Key, CustCount = g.Count() }
```

the final translation of which is

```
customers.
GroupBy(c => c.Country).
Select(g => new { Country = g.Key, CustCount = g.Count() })
```

JOSEPH ALBAHARI The purpose of a query continuation is to allow further clauses after a `select` or `group` clause (which would otherwise terminate the query). After a query continuation, the former range variable, and any variables that were introduced through `join` or `let` clauses, are out of scope. In contrast, a `let` clause acts like a nondestructive `select`: It keeps the former range variable, and other query variables, in scope.

The identifier introduced by a query continuation can be the same as the preceding range variable.

www.it-ebooks.info

7.16.2.2 *Explicit Range Variable Types*

A `from` clause that explicitly specifies a range variable type

```
from T x in e
```

is translated into

```
from x in ( e ) . Cast < T > ( )
```

A `join` clause that explicitly specifies a range variable type

```
join T x in e on k1 equals k2
```

is translated into

```
join x in ( e ) . Cast < T > ( ) on k1 equals k2
```

The translations in the following sections assume that queries have no explicit range variable types.

The example

```
from Customer c in customers
where c.City == "London"
select c
```

is translated into

```
from c in customers.Cast<Customer>()
where c.City == "London"
select c
```

the final translation of which is

```
customers.
Cast<Customer>().
Where(c => c.City == "London")
```

Explicit range variable types are useful for querying collections that implement the non-generic `IEnumerable` interface, but not the generic `IEnumerable<T>` interface. In the example above, this would be the case if `customers` were of type `ArrayList`.

7.16.2.3 *Degenerate Query Expressions*

A query expression of the form

```
from x in e select x
```

is translated into

```
( e ) . Select ( x => x )
```


www.it-ebooks.info

The example

```
from c in customers
select c
```

is translated into

```
customers.Select(c => c)
```

A degenerate query expression is one that trivially selects the elements of the source. A later phase of the translation removes degenerate queries introduced by other translation steps by replacing them with their source. It is important, however, to ensure that the result of a query expression is never the source object itself, as that would reveal the type and identity of the source to the client of the query. Therefore this step protects degenerate queries written directly in source code by explicitly calling `Select` on the source. It is then up to the implementers of `Select` and other query operators to ensure that these methods never return the source object itself.

7.16.2.4 *from, let, where, join, and orderby Clauses*

JOSEPH ALBAHARI The cumbersome-looking translations in this section are what make query syntax really useful: They eliminate the need to write out cumbersome queries by hand. Without this problem, there might have been little justification for introducing query expression syntax into C# 3.0, given the capabilities of lambda expressions and extension methods.

The common theme in the more complex translations is the process of projecting into a temporary anonymous type so as to keep the former range variable in scope following a `let`, `from`, or `join` clause.

A query expression with a second `from` clause followed by a `select` clause

```
from x1 in e1
from x2 in e2
select v
```

is translated into

```
( e1 ) . SelectMany( x1 => e2 , ( x1 , x2 ) => v )
```

A query expression with a second `from` clause followed by something other than a `select` clause

```
from x1 in e1
from x2 in e2
...
```


www.it-ebooks.info

is translated into

```
from * in ( e1 ) . SelectMany( x1 => e2 , ( x1 , x2 ) => new { x1 , x2 } )
```

A query expression with a `let` clause

```
from x in e
let y = f
...
```

is translated into

```
from * in ( e ) . Select ( x => new { x , y = f } )
...
```

A query expression with a `where` clause

```
from x in e
where f
...
```

is translated into

```
from x in ( e ) . Where ( x => f )
...
```

A query expression with a `join` clause without an `into` followed by a `select` clause

```
from x1 in e1
join x2 in e2 on k1 equals k2
select v
```

is translated into

```
( e1 ) . Join( e2 , x1 => k1 , x2 => k2 , ( x1 , x2 ) => v )
```

A query expression with a `join` clause without an `into` followed by something other than a `select` clause

```
from x1 in e1
join x2 in e2 on k1 equals k2
...
```

is translated into

```
from * in ( e1 ) . Join( e2 , x1 => k1 , x2 => k2 , ( x1 , x2 ) => new { x1 , x2 })
...
```


www.it-ebooks.info

A query expression with a `join` clause with an `into` followed by a `select` clause

```
from x1 in e1
join x2 in e2 on k1 equals k2 into g
select v
```

is translated into

```
( e1 ) . GroupJoin( e2 , x1 => k1 , x2 => k2 , ( x1 , g ) => v )
```

A query expression with a `join` clause with an `into` followed by something other than a `select` clause

```
from x1 in e1
join x2 in e2 on k1 equals k2 into g
...
```

is translated into

```
from * in ( e1 ) . GroupJoin( e2 , x1 => k1 , x2 => k2 , ( x1 , g ) => new { x1 , g })
```

A query expression with an `orderby` clause

```
from x in e
orderby k1 , k2 , ... , kn
...
```

is translated into

```
from x in ( e ) .
OrderBy ( x => k1 ) .
ThenBy ( x => k2 ) .
... .
ThenBy ( x => kn )
...
```

If an ordering clause specifies a `descending` direction indicator, an invocation of `OrderByDescending` or `ThenByDescending` is produced instead.

The following translations assume that there are no `let`, `where`, `join`, or `orderby` clauses, and no more than the one initial `from` clause in each query expression.

The example

```
from c in customers
from o in c.Orders
select new { c.Name, o.OrderID, o.Total }
```

is translated into

```
customers.
SelectMany(c => c.Orders, (c,o) => new { c.Name, o.OrderID, o.Total })
```


www.it-ebooks.info

The example

```
from c in customers
from o in c.Orders
orderby o.Total descending
select new { c.Name, o.OrderID, o.Total }
```

is translated into

```
from * in customers.
   SelectMany(c => c.Orders, (c,o) => new { c, o })
orderby o.Total descending
select new { c.Name, o.OrderID, o.Total }
```

the final translation of which is

```
customers.
SelectMany(c => c.Orders, (c,o) => new { c, o }).
OrderByDescending(x => x.o.Total).
Select(x => new { x.c.Name, x.o.OrderID, x.o.Total })
```

where x is a compiler-generated identifier that is otherwise invisible and inaccessible.

The example

```
from o in orders
let t = o.Details.Sum(d => d.UnitPrice * d.Quantity)
where t >= 1000
select new { o.OrderID, Total = t }
```

is translated into

```
from * in orders.
    Select(o => new { o, t = o.Details.Sum(d => d.UnitPrice * d.Quantity) })
where t >= 1000
select new { o.OrderID, Total = t }
```

the final translation of which is

```
orders.
Select(o => new { o, t = o.Details.Sum(d => d.UnitPrice * d.Quantity) }).
Where(x => x.t >= 1000).
Select(x => new { x.o.OrderID, Total = x.t })
```

where x is a compiler-generated identifier that is otherwise invisible and inaccessible.

The example

```
from c in customers
join o in orders on c.CustomerID equals o.CustomerID
select new { c.Name, o.OrderDate, o.Total }
```


www.it-ebooks.info

is translated into

```
customers.Join(orders, c => c.CustomerID, o => o.CustomerID,
    (c, o) => new { c.Name, o.OrderDate, o.Total })
```

The example

```
from c in customers
join o in orders on c.CustomerID equals o.CustomerID into co
let n = co.Count()
where n >= 10
select new { c.Name, OrderCount = n }
```

is translated into

```
from * in customers.
    GroupJoin(orders, c => c.CustomerID, o => o.CustomerID,
        (c, co) => new { c, co })
let n = co.Count()
where n >= 10
select new { c.Name, OrderCount = n }
```

the final translation of which is

```
customers.
GroupJoin(orders, c => c.CustomerID, o => o.CustomerID, (c, co) => new { c, co }).
Select(x => new { x, n = x.co.Count() }).
Where(y => y.n >= 10).
Select(y => new { y.x.c.Name, OrderCount = y.n)
```

where `x` and `y` are compiler-generated identifiers that are otherwise invisible and inaccessible.

The example

```
from o in orders
orderby o.Customer.Name, o.Total descending
select o
```

has the final translation

```
orders.
OrderBy(o => o.Customer.Name).
ThenByDescending(o => o.Total)
```

7.16.2.5 *select Clauses*

A query expression of the form

```
from x in e select v
```

is translated into

www.it-ebooks.info

```
( e ) . Select ( x => v )
```

except when *v* is the identifier *x*, the translation is simply

```
( e )
```

For example,

```
from c in customers.Where(c => c.City == "London")
select c
```

is simply translated into

```
customers.Where(c => c.City == "London")
```

7.16.2.6 *GroupBy Clauses*

A query expression of the form

```
from x in e group v by k
```

is translated into

```
( e ) . GroupBy ( x => k , x => v )
```

except when *v* is the identifier *x*, the translation is

```
( e ) . GroupBy ( x => k )
```

The example

```
from c in customers
group c.Name by c.Country
```

is translated into

```
customers.
GroupBy(c => c.Country, c => c.Name)
```

7.16.2.7 *Transparent Identifiers*

Certain translations inject range variables with ***transparent identifiers*** denoted by *. Transparent identifiers are not a proper language feature; they exist only as an intermediate step in the query expression translation process.

When a query translation injects a transparent identifier, further translation steps propagate the transparent identifier into anonymous functions and anonymous object initializers. In those contexts, transparent identifiers have the following behavior:

- When a transparent identifier occurs as a parameter in an anonymous function, the members of the associated anonymous type are automatically in scope in the body of the anonymous function.

www.it-ebooks.info

- When a member with a transparent identifier is in scope, the members of that member are in scope as well.
- When a transparent identifier occurs as a member declarator in an anonymous object initializer, it introduces a member with a transparent identifier.

In the translation steps described above, transparent identifiers are always introduced together with anonymous types, with the intent of capturing multiple range variables as members of a single object. An implementation of C# is permitted to use a different mechanism than anonymous types to group together multiple range variables. The following translation examples assume that anonymous types are used, and show how transparent identifiers can be translated away.

The example

```
from c in customers
from o in c.Orders
orderby o.Total descending
select new { c.Name, o.Total }
```

is translated into

```
from * in customers.
    SelectMany(c => c.Orders, (c,o) => new { c, o })
orderby o.Total descending
select new { c.Name, o.Total }
```

which is further translated into

```
customers.
SelectMany(c => c.Orders, (c,o) => new { c, o }).
OrderByDescending(* => o.Total).
Select(* => new { c.Name, o.Total })
```

which, when transparent identifiers are erased, is equivalent to

```
customers.
SelectMany(c => c.Orders, (c,o) => new { c, o }).
OrderByDescending(x => x.o.Total).
Select(x => new { x.c.Name, x.o.Total })
```

where x is a compiler-generated identifier that is otherwise invisible and inaccessible.

The example

```
from c in customers
join o in orders on c.CustomerID equals o.CustomerID
join d in details on o.OrderID equals d.OrderID
join p in products on d.ProductID equals p.ProductID
select new { c.Name, o.OrderDate, p.ProductName }
```


www.it-ebooks.info

is translated into

```
from * in customers.
    Join(orders, c => c.CustomerID, o => o.CustomerID, (c, o) => new { c, o })
join d in details on o.OrderID equals d.OrderID
join p in products on d.ProductID equals p.ProductID
select new { c.Name, o.OrderDate, p.ProductName }
```

which is further reduced to

```
customers.
Join(orders, c => c.CustomerID, o => o.CustomerID, (c, o) => new { c, o }).
Join(details, * => o.OrderID, d => d.OrderID, (*, d) => new { *, d }).
Join(products, * => d.ProductID, p => p.ProductID, (*, p) => new { *, p }).
Select(* => new { c.Name, o.OrderDate, p.ProductName })
```

the final translation of which is

```
customers.
Join(orders, c => c.CustomerID, o => o.CustomerID, (c, o) => new { c, o }).
Join(details, x => x.o.OrderID, d => d.OrderID, (x, d) => new { x, d }).
Join(products, y => y.d.ProductID, p => p.ProductID, (y, p) => new { y, p }).
Select(z => new { z.y.x.c.Name, z.y.x.o.OrderDate, z.p.ProductName })
```

where `x`, `y`, and `z` are compiler-generated identifiers that are otherwise invisible and inaccessible.

7.16.3 The Query Expression Pattern

The ***query expression pattern*** establishes a pattern of methods that types can implement to support query expressions. Because query expressions are translated to method invocations by means of a syntactic mapping, types have considerable flexibility in how they implement the query expression pattern. For example, the methods of the pattern can be implemented as instance methods or as extension methods because the two have the same invocation syntax, and the methods can request delegates or expression trees because anonymous functions are convertible to both.

The recommended shape of a generic type `C<T>` that supports the query expression pattern is shown below. A generic type is used to illustrate the proper relationships between parameter and result types, but it is possible to implement the pattern for nongeneric types as well.

```
class C<T> : C
{

    public C<T> Where(Func<T,bool> predicate);

    public C<U> Select<U>(Func<T,U> selector);
```


www.it-ebooks.info

```
    public C<V> SelectMany<U,V>(Func<T,C<U>> selector,
        Func<T,U,V> resultSelector);

    public C<V> Join<U,K,V>(C<U> inner, Func<T,K> outerKeySelector,
        Func<U,K> innerKeySelector, Func<T,U,V> resultSelector);

    public C<V> GroupJoin<U,K,V>(C<U> inner, Func<T,K> outerKeySelector,
        Func<U,K> innerKeySelector, Func<T,C<U>,V> resultSelector);

    public O<T> OrderBy<K>(Func<T,K> keySelector);

    public O<T> OrderByDescending<K>(Func<T,K> keySelector);

    public C<G<K,T>> GroupBy<K>(Func<T,K> keySelector);

    public C<G<K,E>> GroupBy<K,E>(Func<T,K> keySelector,
        Func<T,E> elementSelector);
}

class O<T> : C<T>
{
    public O<T> ThenBy<K>(Func<T,K> keySelector);

    public O<T> ThenByDescending<K>(Func<T,K> keySelector);
}

class G<K,T> : C<T>
{
    public K Key { get; }
}
```

The methods above use the generic delegate types `Func<T1, R>` and `Func<T1, T2, R>`, but they could equally well have used other delegate or expression tree types with the same relationships in parameter and result types.

Notice the recommended relationship between `C<T>` and `O<T>`, which ensures that the `ThenBy` and `ThenByDescending` methods are available only on the result of an `OrderBy` or `OrderByDescending`. Also notice the recommended shape of the result of `GroupBy`—a sequence of sequences, where each inner sequence has an additional `Key` property.

> **BILL WAGNER** `ThenBy` will often have better performance than `OrderBy`, because it needs to sort only inner sequences that have more than one value.

The `System.Linq` namespace provides an implementation of the query operator pattern for any type that implements the `System.Collections.Generic.IEnumerable<T>` interface.

> **BILL WAGNER** There is also an implementation for any type that implements `IQueryable<T>`.

www.it-ebooks.info

ERIC LIPPERT This signature for `Join` is one of the primary motivators of the "accumulate bounds and then fix to the best one" part of the method type inference algorithm. If the inner key is, say, of type `int`, and the outer key of type `int?`, then rather than having type inference fail due to the "contradiction," it is better to simply pick the more general of the two types. Because every `int` is an `int?`, the type inference algorithm would choose `int?` for `K`.

7.17 Assignment Operators

The assignment operators assign a new value to a variable, a property, an event, or an indexer element.

assignment:
unary-expression *assignment-operator* *expression*

assignment-operator:
`=`
`+=`
`-=`
`*=`
`/=`
`%=`
`&=`
`|=`
`^=`
`<<=`
right-shift-assignment

The left operand of an assignment must be an expression classified as a variable, a property access, an indexer access, or an event access.

The `=` operator is called the ***simple assignment operator***. It assigns the value of the right operand to the variable, property, or indexer element given by the left operand. The left operand of the simple assignment operator may not be an event access (except as described in §10.8.1). The simple assignment operator is described in §7.17.1.

The assignment operators other than the `=` operator are called the ***compound assignment operators***. These operators perform the indicated operation on the two operands, and then assign the resulting value to the variable, property, or indexer element given by the left operand. The compound assignment operators are described in §7.17.2.

www.it-ebooks.info

The += and -= operators with an event access expression as the left operand are called the *event assignment operators*. No other assignment operator is valid with an event access as the left operand. The event assignment operators are described in §7.17.3.

The assignment operators are right-associative, meaning that operations are grouped from right to left. For example, an expression of the form a = b = c is evaluated as a = (b = c).

7.17.1 Simple Assignment

The = operator is called the simple assignment operator.

If the left operand of a simple assignment is of the form E.P or E[E_i], where E has the compile-time type dynamic, then the assignment is dynamically bound (§7.2.2). In this case, the compile-time type of the assignment expression is dynamic, and the resolution described below will take place at runtime based on the runtime type of E.

In a simple assignment, the right operand must be an expression that is implicitly convertible to the type of the left operand. The operation assigns the value of the right operand to the variable, property, or indexer element given by the left operand.

The result of a simple assignment expression is the value assigned to the left operand. The result has the same type as the left operand and is always classified as a value.

If the left operand is a property or indexer access, the property or indexer must have a set accessor. If this is not the case, a binding-time error occurs.

The runtime processing of a simple assignment of the form x = y consists of the following steps:

- If x is classified as a variable:
  - x is evaluated to produce the variable.
  - y is evaluated and, if required, converted to the type of x through an implicit conversion (§6.1).
  - If the variable given by x is an array element of a *reference-type*, a runtime check is performed to ensure that the value computed for y is compatible with the array instance of which x is an element. The check succeeds if y is null, or if an implicit reference conversion (§6.1.6) exists from the actual type of the instance referenced by y to the actual element type of the array instance containing x. Otherwise, a System.ArrayTypeMismatchException is thrown.
  - The value resulting from the evaluation and conversion of y is stored into the location given by the evaluation of x.

www.it-ebooks.info

- If x is classified as a property or indexer access:
  - The instance expression (if x is not `static`) and the argument list (if x is an indexer access) associated with x are evaluated, and the results are used in the subsequent `set` accessor invocation.
  - y is evaluated and, if required, converted to the type of x through an implicit conversion (§6.1).
  - The `set` accessor of x is invoked with the value computed for y as its `value` argument.

The array covariance rules (§12.5) permit a value of an array type `A[]` to be a reference to an instance of an array type `B[]`, provided an implicit reference conversion exists from `B` to `A`. Because of these rules, assignment to an array element of a *reference-type* requires a run-time check to ensure that the value being assigned is compatible with the array instance. In the example

```
string[] sa = new string[10];
object[] oa = sa;

oa[0] = null;               // Okay
oa[1] = "Hello";            // Okay
oa[2] = new ArrayList();    // ArrayTypeMismatchException
```

the last assignment causes a `System.ArrayTypeMismatchException` to be thrown because an instance of `ArrayList` cannot be stored in an element of a `string[]`.

BILL WAGNER This point implies that array assignment does not copy the array, but rather adds a new reference to the same storage.

When a property or indexer declared in a *struct-type* is the target of an assignment, the instance expression associated with the property or indexer access must be classified as a variable. If the instance expression is classified as a value, a binding-time error occurs. Because of §7.6.4, the same rule also applies to fields.

Given the declarations:

```
struct Point
{
    int x, y;

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


www.it-ebooks.info

```
        public int X
        {
            get { return x; }
            set { x = value; }
        }

        public int Y
        {
            get { return y; }
            set { y = value; }
        }
    }

    struct Rectangle
    {
        Point a, b;

        public Rectangle(Point a, Point b)
        {
            this.a = a;
            this.b = b;
        }

        public Point A
        {
            get { return a; }
            set { a = value; }
        }

        public Point B
        {
            get { return b; }
            set { b = value; }
        }
    }
```

in the example

```
Point p = new Point();
p.X = 100;
p.Y = 100;
Rectangle r = new Rectangle();
r.A = new Point(10, 10);
r.B = p;
```

the assignments to `p.X`, `p.Y`, `r.A`, and `r.B` are permitted because `p` and `r` are variables. However, in the example

```
Rectangle r = new Rectangle();
r.A.X = 10;
r.A.Y = 10;
r.B.X = 100;
r.B.Y = 100;
```

the assignments are all invalid because `r.A` and `r.B` are not variables.

www.it-ebooks.info

JOSEPH ALBAHARI An early release of the C# 1.0 compiler allowed assignments such as `r.A.X = 10`—but they failed silently because `r.A` returns a *copy* of a `Point` (i.e., a value) rather than a variable. People found this behavior confusing, so the condition was detected and reported as an error.

BILL WAGNER This discussion highlights yet another reason why structs should be immutable.

7.17.2 Compound Assignment

If the left operand of a compound assignment is of the form `E.P` or `E[E`$_i$`]`, where `E` has the compile-time type `dynamic`, then the assignment is dynamically bound (§7.2.2). In this case, the compile-time type of the assignment expression is `dynamic`, and the resolution described below will take place at runtime based on the runtime type of `E`.

An operation of the form `x` *op*`=` `y` is processed by applying binary operator overload resolution (§7.3.4) as if the operation was written `x` *op* `y`. Then,

- If the return type of the selected operator is *implicitly* convertible to the type of `x`, the operation is evaluated as `x = x` *op* `y`, except that `x` is evaluated only once.
- Otherwise, if the selected operator is a predefined operator, if the return type of the selected operator is *explicitly* convertible to the type of `x`, and if `y` is *implicitly* convertible to the type of `x` or the operator is a shift operator, then the operation is evaluated as `x = (T)(x` *op* `y)`, where `T` is the type of `x`, except that `x` is evaluated only once.
- Otherwise, the compound assignment is invalid, and a binding-time error occurs.

PETER SESTOFT The condition of `y` being implicitly convertible to the type of `x` in the second item means that the compound operators are more restrictive in C# than in the C, C++, and Java programming languages. In those languages, the assignment `x+=0.9` is legal even if variable `x` has integer type. When `x` is non-negative, the assignment has no effect; when `x` is negative, it adds 1 to `x`. In C#, the assignment is rejected—wisely, in my opinion.

VLADIMIR RESHETNIKOV A curious consequence of these rules is that the expression `x <<= null` is legal if the variable `x` has type `int`. Of course, it will always throw an exception at runtime.

www.it-ebooks.info

The term "evaluated only once" means that in the evaluation of `x` *op* `y`, the results of any constituent expressions of `x` are temporarily saved and then reused when performing the assignment to `x`. For example, in the assignment `A()[B()] += C()`, where `A` is a method returning `int[]`, and `B` and `C` are methods returning `int`, the methods are invoked only once, in the order `A`, `B`, `C`.

When the left operand of a compound assignment is a property access or an indexer access, the property or indexer must have both a `get` accessor and a `set` accessor. If this is not the case, a binding-time error occurs.

The second rule above permits `x` *op*`= y` to be evaluated as `x = (T)(x` *op* `y)` in certain contexts. The rule exists such that the predefined operators can be used as compound operators when the left operand is of type `sbyte`, `byte`, `short`, `ushort`, or `char`. Even when both arguments are of one of those types, the predefined operators produce a result of type `int`, as described in §7.3.6.2. Thus, without a cast, it would not be possible to assign the result to the left operand.

The intuitive effect of the rule for predefined operators is simply that `x` *op*`= y` is permitted if both `x` *op* `y` and `x = y` are permitted. In the example

```
byte b = 0;
char ch = '\0';
int i = 0;

b += 1;                 // Okay
b += 1000;              // Error: b = 1000 not permitted
b += i;                 // Error: b = i not permitted
b += (byte)i;           // Okay

ch += 1;                // Error: ch = 1 not permitted
ch += (char)1;          // Okay
```

the intuitive reason for each error is that a corresponding simple assignment would also have been an error.

This also means that compound assignment operations support lifted operations. In the example

```
int? i = 0;
i += 1;                 // Ok
```

the lifted operator `+(int?,int?)` is used.

7.17.3 Event Assignment

If the left operand of a `+=` or `-=` operator is classified as an event access, then the expression is evaluated as follows:

www.it-ebooks.info

- The instance expression, if any, of the event access is evaluated.
- The right operand of the `+=` or `-=` operator is evaluated, and, if required, converted to the type of the left operand through an implicit conversion (§6.1).
- An event accessor of the event is invoked, with argument list consisting of the right operand, after evaluation and, if necessary, conversion. If the operator was `+=`, the `add` accessor is invoked; if the operator was `-=`, the `remove` accessor is invoked.

An event assignment expression does not yield a value. Thus an event assignment expression is valid only in the context of a *statement-expression* (§8.6).

7.18 Expression

An *expression* is either a *non-assignment-expression* or an *assignment*.

expression:
    non-assignment-expression
    assignment

non-assignment-expression:
    conditional-expression
    lambda-expression
    query-expression

7.19 Constant Expressions

A *constant-expression* is an expression that can be fully evaluated at compile time.

constant-expression:
    expression

A constant expression must be the `null` literal or a value with one of the following types: `sbyte`, `byte`, `short`, `ushort`, `int`, `uint`, `long`, `ulong`, `char`, `float`, `double`, `decimal`, `bool`, `string`, or any enumeration type. Only the following constructs are permitted in constant expressions:

- Literals (including the `null` literal).
- References to `const` members of class and struct types.
- References to members of enumeration types.
- References to `const` parameters or local variables.
- Parenthesized subexpressions, which are themselves constant expressions.

www.it-ebooks.info

- Cast expressions, provided the target type is one of the types listed above.
- `checked` and `unchecked` expressions.
- Default value expressions.
- The predefined `+`, `-`, `!`, and `~` unary operators.
- The predefined `+`, `-`, `*`, `/`, `%`, `<<`, `>>`, `&`, `|`, `^`, `&&`, `||`, `==`, `!=`, `<`, `>`, `<=`, and `>=` binary operators, provided each operand is of a type listed above.
- The `?:` conditional operator.

The following conversions are permitted in constant expressions:

- Identity conversions.
- Numeric conversions.
- Enumeration conversions.
- Constant expression conversions.
- Implicit and explicit reference conversions, provided that the source of the conversions is a constant expression that evaluates to the `null` value.

Other conversions, including boxing, unboxing, and implicit reference conversions of non-null values, are not permitted in constant expressions. For example:

```
class C {
    const object i = 5;          // Error: boxing conversion not permitted
    const object str = "hello";  // Error: implicit reference conversion
}
```

In this example, the initialization of `i` is an error because a boxing conversion is required. The initialization of `str` is an error because an implicit reference conversion from a non-null value is required.

Whenever an expression fulfills the requirements listed above, the expression is evaluated at compile time. This is true even if the expression is a subexpression of a larger expression that contains nonconstant constructs.

The compile-time evaluation of constant expressions uses the same rules as runtime evaluation of nonconstant expressions, except that where runtime evaluation would have thrown an exception, compile-time evaluation causes a compile-time error to occur.

Unless a constant expression is explicitly placed in an `unchecked` context, overflows that occur in integral-type arithmetic operations and conversions during the compile-time evaluation of the expression always cause compile-time errors (§7.19).

www.it-ebooks.info

Constant expressions occur in the contexts listed below. In these contexts, a compile-time error occurs if an expression cannot be fully evaluated at compile time.

- Constant declarations (§10.4).
- Enumeration member declarations (§14.3).
- `case` labels of a `switch` statement (§8.7.2).
- `goto case` statements (§8.9.3).
- Dimension lengths in an array creation expression (§7.6.10.4) that includes an initializer.
- Attributes (§17).

An implicit constant expression conversion (§6.1.8) permits a constant expression of type `int` to be converted to `sbyte`, `byte`, `short`, `ushort`, `uint`, or `ulong`, provided the value of the constant expression is within the range of the destination type.

7.20 Boolean Expressions

A *boolean-expression* is an expression that yields a result of type `bool`, either directly or through application of `operator true` in certain contexts as specified in the following.

boolean-expression:
expression

The controlling conditional expression of an *if-statement* (§8.7.1), *while-statement* (§8.8.1), *do-statement* (§8.8.2), or *for-statement* (§8.8.3) is a *boolean-expression*. The controlling conditional expression of the `?:` operator (§7.14) follows the same rules as a *boolean-expression*, but for reasons of operator precedence is classified as a *conditional-or-expression*.

A *boolean-expression* is required to be implicitly convertible to `bool` or of a type that implements `operator true`. If neither requirement is satisfied, a binding-time error occurs.

When a boolean expression cannot be implicitly converted to `bool` but does implement `operator true`, then following evaluation of the expression, the `operator true` implementation provided by that type is invoked to produce a `bool` value.

The `DBBool` struct type in §11.4.2 provides an example of a type that implements `operator true` and `operator false`.

www.it-ebooks.info

This page intentionally left blank

www.it-ebooks.info

8. Statements

C# provides a variety of statements. Most of these statements will be familiar to developers who have programmed in C and C++.

statement:
    labeled-statement
    declaration-statement
    embedded-statement

embedded-statement:
    block
    empty-statement
    expression-statement
    selection-statement
    iteration-statement
    jump-statement
    try-statement
    checked-statement
    unchecked-statement
    lock-statement
    using-statement
    yield-statement

The *embedded-statement* nonterminal is used for statements that appear within other statements. The use of *embedded-statement* rather than *statement* excludes the use of declaration statements and labeled statements in these contexts. The example

```
void F(bool b) {
    if (b)
        int i = 44;
}
```

results in a compile-time error because an `if` statement requires an *embedded-statement* rather than a *statement* for its `if` branch. If this code were permitted, then the variable `i`

www.it-ebooks.info

would be declared, but it could never be used. Note, however, that by placing `i`'s declaration in a block, the example is valid.

8.1 End Points and Reachability

Every statement has an ***end point***. In intuitive terms, the end point of a statement is the location that immediately follows the statement. The execution rules for composite statements (statements that contain embedded statements) specify the action that is taken when control reaches the end point of an embedded statement. For example, when control reaches the end point of a statement in a block, control is transferred to the next statement in the block.

If a statement can possibly be reached by execution, the statement is said to be ***reachable***. Conversely, if there is no possibility that a statement will be executed, the statement is said to be ***unreachable***.

In the example

```
void F() {
    Console.WriteLine("reachable");
    goto Label;
    Console.WriteLine("unreachable");
    Label:
    Console.WriteLine("reachable");
}
```

the second invocation of `Console.WriteLine` is unreachable because there is no possibility that the statement will be executed.

A warning is reported if the compiler determines that a statement is unreachable. It is specifically not an error for a statement to be unreachable.

To determine whether a particular statement or end point is reachable, the compiler performs flow analysis according to the reachability rules defined for each statement. The flow analysis takes into account the values of constant expressions (§7.19) that control the behavior of statements, but the possible values of nonconstant expressions are not considered. In other words, for purposes of control flow analysis, a nonconstant expression of a given type is considered to have any possible value of that type.

In the example

```
void F() {
    const int i = 1;
    if (i == 2) Console.WriteLine("unreachable");
}
```


www.it-ebooks.info

the boolean expression of the `if` statement is a constant expression because both operands of the `==` operator are constants. As the constant expression is evaluated at compile time, producing the value `false`, the `Console.WriteLine` invocation is considered unreachable. However, if `i` is changed to be a local variable

```
void F() {
    int i = 1;
    if (i == 2) Console.WriteLine("reachable");
}
```

the `Console.WriteLine` invocation is considered reachable, even though, in reality, it will never be executed.

> **ERIC LIPPERT** Many other expressions that humans know will always be false are not considered to be false by the flow analyzer. For example, if we replaced the condition of the `if` statement above with `(i * 0 == 0)`, then the consequence would be reachable according to the specification—even though you and I know that it will not be reachable in practice.

The *block* of a function member is always considered reachable. By successively evaluating the reachability rules of each statement in a block, the reachability of any given statement can be determined.

In the example

```
void F(int x) {
    Console.WriteLine("start");
    if (x < 0) Console.WriteLine("negative");
}
```

the reachability of the second `Console.WriteLine` is determined as follows:

- The first `Console.WriteLine` expression statement is reachable because the block of the `F` method is reachable.
- The end point of the first `Console.WriteLine` expression statement is reachable because that statement is reachable.
- The `if` statement is reachable because the end point of the first `Console.WriteLine` expression statement is reachable.
- The second `Console.WriteLine` expression statement is reachable because the boolean expression of the `if` statement does not have the constant value `false`.

www.it-ebooks.info

There are two situations in which it is a compile-time error for the end point of a statement to be reachable:

- Because the `switch` statement does not permit a switch section to "fall through" to the next switch section, it is a compile-time error for the end point of the statement list of a switch section to be reachable. If this error occurs, it is typically an indication that a `break` statement is missing.
- It is a compile-time error for the end point of the block of a function member that computes a value to be reachable. If this error occurs, it typically is an indication that a `return` statement is missing.

BILL WAGNER The rules for reachable code are designed to err on the side of assuming code is reachable. For example, the following example does not generate any warnings for unreachable code:

```
public class Program
{
    public static int counter = 5;

    static void Main(string[] args)
    {
        if (counter == 6)
            Console.WriteLine("weird");
        else
            Console.WriteLine("normal");
    }
}
```

It's obvious to human readers that "weird" will never appear on your console. The language rules do not follow the same reasoning.

8.2 Blocks

A *block* permits multiple statements to be written in contexts where a single statement is allowed.

block:
{ *statement-list*$_{opt}$ }

A *block* consists of an optional *statement-list* (§8.2.1), enclosed in braces. If the statement list is omitted, the block is said to be empty.

www.it-ebooks.info

A block may contain declaration statements (§8.5). The scope of a local variable or constant declared in a block is the block.

Within a block, the meaning of a name used in an expression context must always be the same (§7.6.2.1).

A block is executed as follows:

- If the block is empty, control is transferred to the end point of the block.
- If the block is not empty, control is transferred to the statement list. When and if control reaches the end point of the statement list, control is transferred to the end point of the block.

The statement list of a block is reachable if the block itself is reachable.

The end point of a block is reachable if the block is empty or if the end point of the statement list is reachable.

A *block* that contains one or more `yield` statements (§8.14) is called an iterator block. Iterator blocks are used to implement function members as iterators (§10.14). Some additional restrictions apply to iterator blocks:

- It is a compile-time error for a `return` statement to appear in an iterator block (but `yield return` statements are permitted).
- It is a compile-time error for an iterator block to contain an unsafe context (§18.1). An iterator block always defines a safe context, even when its declaration is nested in an unsafe context.

8.2.1 Statement Lists

A ***statement list*** consists of one or more statements written in sequence. Statement lists occur in *block*s (§8.2) and in *switch-block*s (§8.7.2).

statement-list:
    statement
    statement-list statement

A statement list is executed by transferring control to the first statement. When and if control reaches the end point of a statement, control is transferred to the next statement. When and if control reaches the end point of the last statement, control is transferred to the end point of the statement list.

www.it-ebooks.info

A statement in a statement list is reachable if at least one of the following is true:

- The statement is the first statement and the statement list itself is reachable.
- The end point of the preceding statement is reachable.
- The statement is a labeled statement and the label is referenced by a reachable `goto` statement.

VLADIMIR RESHETNIKOV This rule is not applied when the `goto` statement is placed inside a `try` or `catch` block of a `try` statement that includes a `finally` block, and the labeled statement is outside the `try` statement, and the end point of the `finally` block is unreachable. For example:

```
    class C
    {
        static void Main()
        {
            int x;
            try
            {
                goto A; // Reachable statement
            }
            finally
            {
                throw new System.Exception();
            }
            A: x.ToString(); // Unreachable statement
        }
}
```

The end point of a statement list is reachable if the end point of the last statement in the list is reachable.

8.3 The Empty Statement

An *empty-statement* does nothing.

empty-statement:
    ;

An empty statement is used when there are no operations to perform in a context where a statement is required.

www.it-ebooks.info

JON SKEET Given how easy it is to miss an empty statement when reading code, I wonder whether something larger might be useful for this relatively rare requirement. For example, the following code is legal but probably not what's intended:

```
while (text.IndexOf("xx") != -1);
{
    text = text.Replace("xx", "x");
}
```

The empty statement for the `while` loop is fairly well camouflaged. If you had to write something like "`void;`" when you actually wanted an empty statement, the compiler could flag "accidentally empty" statements like the previous example as errors.

JESSE LIBERTY Jon's example shows why empty statements should be rare, and called out (though not necessarily with comments). The problem in his example code can be rectified by following the best practice of empty statements existing either on a line by themselves or, even better, within full braces:

```
while (text.IndexOf("xx") != -1)
{
   ;
}



{
    text = text.Replace("xx", "x");
}
```

No programmer can now miss that the `while` loop has an empty statement. The purpose of the next set of braces is no more (or less) ambiguous than it was, but the white space helped avoid the confusion.

Execution of an empty statement simply transfers control to the end point of the statement. Thus the end point of an empty statement is reachable if the empty statement is reachable.

An empty statement can be used when writing a `while` statement with a null body:

```
bool ProcessMessage() {...}

void ProcessMessages() {
    while (ProcessMessage())
        ;
}
```


www.it-ebooks.info

Also, an empty statement can be used to declare a label just before the closing "}" of a block:

```
void F() {
    ...
    if (done) goto exit;
    ...
    exit: ;
}
```

8.4 Labeled Statements

A *labeled-statement* permits a statement to be prefixed by a label. Labeled statements are permitted in blocks, but are not permitted as embedded statements.

labeled-statement:
    identifier `:` *statement*

A labeled statement declares a label with the name given by the *identifier*. The scope of a label is the whole block in which the label is declared, including any nested blocks. It is a compile-time error for two labels with the same name to have overlapping scopes.

A label can be referenced from `goto` statements (§8.9.3) within the scope of the label. This means that `goto` statements can transfer control within blocks and out of blocks, but never into blocks.

Labels have their own declaration space and do not interfere with other identifiers. The example

```
int F(int x) {
    if (x >= 0) goto x;
    x = -x;
    x: return x;
}
```

is valid and uses the name `x` as both a parameter and a label.

Execution of a labeled statement corresponds exactly to execution of the statement following the label.

In addition to the reachability provided by normal flow of control, a labeled statement is reachable if the label is referenced by a reachable `goto` statement. (Exception: If a `goto` statement is inside a `try` that includes a `finally` block, and the labeled statement is outside the `try`, and the end point of the `finally` block is unreachable, then the labeled statement is not reachable from that `goto` statement.)

www.it-ebooks.info

ERIC LIPPERT For example, the `finally` block might always throw an exception, in which case there would be a reachable `goto` targeting a potentially unreachable label.

PETER SESTOFT See also the annotations in the discussion of the `goto` statement in §8.9.3.

8.5 Declaration Statements

A *declaration-statement* declares a local variable or constant. Declaration statements are permitted in blocks, but are not permitted as embedded statements.

declaration-statement:
    local-variable-declaration `;`
    local-constant-declaration `;`

8.5.1 Local Variable Declarations

A *local-variable-declaration* declares one or more local variables.

local-variable-declaration:
    local-variable-type *local-variable-declarators*

local-variable-type:
    type
    `var`

local-variable-declarators:
    local-variable-declarator
    local-variable-declarators `,` *local-variable-declarator*

local-variable-declarator:
    identifier
    identifier `=` *local-variable-initializer*

local-variable-initializer:
    expression
    array-initializer

www.it-ebooks.info

The *local-variable-type* of a *local-variable-declaration* either directly specifies the type of the variables introduced by the declaration, or indicates with the identifier `var` that the type should be inferred based on an initializer. The type is followed by a list of *local-variable-declarator*s, each of which introduces a new variable. A *local-variable-declarator* consists of an *identifier* that names the variable, optionally followed by an "=" token and a *local-variable-initializer* that gives the initial value of the variable.

In the context of a local variable declaration, the identifier `var` acts as a contextual keyword (§2.4.3).When the *local-variable-type* is specified as `var` and no type named `var` is in scope, the declaration is an ***implicitly typed local variable declaration***, whose type is inferred from the type of the associated initializer expression. Implicitly typed local variable declarations are subject to the following restrictions:

- The *local-variable-declaration* cannot include multiple *local-variable-declarator*s.
- The *local-variable-declarator* must include a *local-variable-initializer*.
- The *local-variable-initializer* must be an *expression*.
- The initializer *expression* must have a compile-time type.
- The initializer *expression* cannot refer to the declared variable itself.

ERIC LIPPERT In an early design for this feature, it was legal to have multiple declarators.

```
var a = 1, b = 2.5;
```

When C# developers were shown this code, roughly half said that it should have the same semantics as

```
double a = 1, b = 2.5;
```

The other half said that it should have the same semantics as

```
int a = 1; double b = 2.5;
```

Both sides thought that their interpretation was the "obviously correct" one.

When faced with a syntax that admits two incompatible "obviously correct" interpretations, often the best thing to do is to disallow the syntax entirely rather than to breed confusion.

CHRIS SELLS I think multiple variable declarations in the same statement just to reuse the type name should be illegal, too.

www.it-ebooks.info

ERIC LIPPERT This constraint stands in contrast to an explicitly typed local variable declaration, which does permit an initializer to reference itself.

For example, `int j = M(out j);` is strange, but legal. If this expression were `var j = M(out j)`, then overload resolution could not determine the type returned by `M`, and hence the type of `j`, until the type of the argument was known. Of course, the type of the argument is exactly what we are trying to determine.

Rather than attempting to solve this "chicken and egg" problem, the language specification simply makes this case illegal.

The following are examples of incorrect implicitly typed local variable declarations:

```
var x;              // Error: no initializer to infer type from
var y = {1, 2, 3};  // Error: array initializer not permitted
var z = null;       // Error: null does not have a type
var u = x => x + 1; // Error: anonymous functions do not have a type
var v = v++;        // Error: initializer cannot refer to variable itself
```

The value of a local variable is obtained in an expression using a *simple-name* (§7.6.2), and the value of a local variable is modified using an *assignment* (§7.17). A local variable must be definitely assigned (§5.3) at each location where its value is obtained.

CHRIS SELLS I really love implicitly typed local variable declarations when the type is anonymous (in which case, you have to use them) or when the type of the variable is made explicit as part of the statement that initializes it, but not because you're too lazy to type!

For example:

```
var a = new { Name = "Bob", Age = 42 }; // Good
var b = 1;                              // Good
var v = new Person();                   // Good
var d = GetPerson();                    // BAD!
```

The compiler is perfectly happy with `d` as an implicitly typed variable, but pity the poor human reader!

www.it-ebooks.info

ERIC LIPPERT I generally agree with the annotation above but would make one additional point: `var` works well when the writer of the code is attempting to communicate "the storage type of this variable is unimportant; what is important is the meaning of the variable, not its implementation details." For example, I'll often write code like "`var attributes = ParseMethodAttributes();`"—I am saying here that it doesn't matter whether what comes back is `AttributeSyntax[]` or `List<AttributeSyntax>`. What matters is that the collection of attributes has been parsed.

BILL WAGNER I freely admit to being guilty of using `var` extensively. In fact, my habit is to use implicitly typed variables for almost everything except simple types. I find that it's much more important to have a semantic understanding of a variable rather than a syntactic understanding of a variable. In Chris's example, `GetPerson()` would logically return a `Person`, or an `IPerson`, or something derived from `Person` or from implementing `IPerson`. In all those cases, I'm fine with that bit of ambiguity. I understand the concept of "a variable that represents something person-like" without knowing its exact type.

The scope of a local variable declared in a *local-variable-declaration* is the block in which the declaration occurs. It is an error to refer to a local variable in a textual position that precedes the *local-variable-declarator* of the local variable. Within the scope of a local variable, it is a compile-time error to declare another local variable or constant with the same name.

A local variable declaration that declares multiple variables is equivalent to multiple declarations of single variables with the same type. Furthermore, a variable initializer in a local variable declaration corresponds exactly to an assignment statement that is inserted immediately after the declaration.

The example

```
void F() {
    int x = 1, y, z = x * 2;
}
```

corresponds exactly to

```
void F() {
    int x; x = 1;
    int y;
    int z; z = x * 2;
}
```


www.it-ebooks.info

In an implicitly typed local variable declaration, the type of the local variable being declared is taken to be the same as the type of the expression used to initialize the variable. For example:

```
var i = 5;
var s = "Hello";
var d = 1.0;
var numbers = new int[] {1, 2, 3};
var orders = new Dictionary<int,Order>();
```

The implicitly typed local variable declarations above are precisely equivalent to the following explicitly typed declarations:

```
int i = 5;
string s = "Hello";
double d = 1.0;
int[] numbers = new int[] {1, 2, 3};
Dictionary<int,Order> orders = new Dictionary<int,Order>();
```

PETER SESTOFT Whereas it is possible to declare local (compile-time) constants, one cannot declare a read-only local variable or read-only method parameter in C#, unlike Standard ML's and Scala's `val`, F#'s `let`, and Java's `final`. Being an old functional programmer, I find this a bit sad; often I just want to name a value, not declare a mutable variable. The `using` statement (§8.13) does allow me to declare an immutable local variable, but that's cumbersome and unidiomatic (and hence confusing to most C# developers), and it works only for local variables, not for method parameters.

8.5.2 Local Constant Declarations

A *local-constant-declaration* declares one or more local constants.

local-constant-declaration:
`const` *type* *constant-declarators*

constant-declarators:
constant-declarator
constant-declarators `,` *constant-declarator*

constant-declarator:
identifier `=` *constant-expression*

The *type* of a *local-constant-declaration* specifies the type of the constants introduced by the declaration. The type is followed by a list of *constant-declarator*s, each of which introduces

www.it-ebooks.info

a new constant. A *constant-declarator* consists of an *identifier* that names the constant, followed by an "=" token, followed by a *constant-expression* (§7.19) that gives the value of the constant.

The *type* and *constant-expression* of a local constant declaration must follow the same rules as those of a constant member declaration (§10.4).

The value of a local constant is obtained in an expression using a *simple-name* (§7.6.2).

The scope of a local constant is the block in which the declaration occurs. It is an error to refer to a local constant in a textual position that precedes its *constant-declarator*. Within the scope of a local constant, it is a compile-time error to declare another local variable or constant with the same name.

A local constant declaration that declares multiple constants is equivalent to multiple declarations of single constants with the same type.

8.6 Expression Statements

An *expression-statement* evaluates a given expression. The value computed by the expression, if any, is discarded.

expression-statement:
    statement-expression `;`

statement-expression:
    invocation-expression
    object-creation-expression
    assignment
    post-increment-expression
    post-decrement-expression
    pre-increment-expression
    pre-decrement-expression

Not all expressions are permitted as statements. In particular, expressions such as `x + y` and `x == 1` that merely compute a value (which will be discarded) are not permitted as statements.

Execution of an *expression-statement* evaluates the contained expression and then transfers control to the end point of the *expression-statement*. The end point of an *expression-statement* is reachable if that *expression-statement* is reachable.

www.it-ebooks.info

JON SKEET Occasionally, someone suggests that methods should be allowed to declare themselves as never-ending; that is, they never return normally. With this scheme, the only way in which they could terminate would be via an exception. An obvious example of such a method would be `Assert.Fail` in a unit testing library.

If such a feature ever appeared, the end point of expression statements that invoked such methods would be unreachable. Just occasionally, this syntax would avoid the need for a "dummy" return or throw statement that you know will never be executed. I suspect the extra complexity (in terms of both the language and the implementation) is greater than the slight benefit afforded.

ERIC LIPPERT The committee that standardizes the ECMAScript language has in the past proposed a "never" type that has the semantics Jon is describing. I agree that it would be nice to have in C#, but at this point the cost probably outweighs the relatively small benefits.

8.7 Selection Statements

Selection statements select one of a number of possible statements for execution based on the value of some expression.

selection-statement:
if-statement
switch-statement

8.7.1 The if Statement

The `if` statement selects a statement for execution based on the value of a boolean expression.

if-statement:
`if` `(` *boolean-expression* `)` *embedded-statement*
`if` `(` *boolean-expression* `)` *embedded-statement* `else` *embedded-statement*

An `else` part is associated with the lexically nearest preceding `if` that is allowed by the syntax. Thus an `if` statement of the form

```
if (x) if (y) F(); else G();
```


www.it-ebooks.info

is equivalent to

```
if (x) {
    if (y) {
        F();
    }
    else {
        G();
    }
}
```

An `if` statement is executed as follows:

- The *boolean-expression* (§7.20) is evaluated.
- If the boolean expression yields `true`, control is transferred to the first embedded statement. When and if control reaches the end point of that statement, control is transferred to the end point of the `if` statement.
- If the boolean expression yields `false` and if an `else` part is present, control is transferred to the second embedded statement. When and if control reaches the end point of that statement, control is transferred to the end point of the `if` statement.
- If the boolean expression yields `false` and if an `else` part is not present, control is transferred to the end point of the `if` statement.

The first embedded statement of an `if` statement is reachable if the `if` statement is reachable and the boolean expression does not have the constant value `false`.

The second embedded statement of an `if` statement, if present, is reachable if the `if` statement is reachable and the boolean expression does not have the constant value `true`.

The end point of an `if` statement is reachable if the end point of at least one of its embedded statements is reachable. In addition, the end point of an `if` statement with no `else` part is reachable if the `if` statement is reachable and the boolean expression does not have the constant value `true`.

8.7.2 The switch Statement

The `switch` statement selects for execution a statement list having an associated switch label that corresponds to the value of the switch expression.

switch-statement:
`switch` `(` *expression* `)` *switch-block*

switch-block:
`{` *switch-sections*$_{opt}$ `}`

www.it-ebooks.info

switch-sections:
    switch-section
    switch-sections *switch-section*

switch-section:
    switch-labels *statement-list*

switch-labels:
    switch-label
    switch-labels *switch-label*

switch-label:
    `case` *constant-expression* `:`
    `default` `:`

A *switch-statement* consists of the keyword `switch`, followed by a parenthesized expression (called the switch expression), followed by a *switch-block*. The *switch-block* consists of zero or more *switch-section*s, enclosed in braces. Each *switch-section* consists of one or more *switch-labels* followed by a *statement-list* (§8.2.1).

The ***governing type*** of a `switch` statement is established by the switch expression.

- If the type of the switch expression is `sbyte`, `byte`, `short`, `ushort`, `int`, `uint`, `long`, `ulong`, `bool`, `char`, `string`, or an *enum-type*, or if it is the nullable type corresponding to one of these types, then that is the governing type of the `switch` statement.
- Otherwise, exactly one user-defined implicit conversion (§6.4) must exist from the type of the switch expression to one of the following possible governing types: `sbyte`, `byte`, `short`, `ushort`, `int`, `uint`, `long`, `ulong`, `char`, `string`, or, a nullable type corresponding to one of those types.
- Otherwise, if no such implicit conversion exists, or if more than one such implicit conversion exists, a compile-time error occurs.

The constant expression of each `case` label must denote a value that is implicitly convertible (§6.1) to the governing type of the `switch` statement. A compile-time error occurs if two or more `case` labels in the same `switch` statement specify the same constant value.

There can be at most one `default` label in a `switch` statement.

A `switch` statement is executed as follows:

- The switch expression is evaluated and converted to the governing type.
- If one of the constants specified in a `case` label in the same `switch` statement is equal to the value of the switch expression, control is transferred to the statement list following the matched `case` label.

www.it-ebooks.info

- If none of the constants specified in `case` labels in the same `switch` statement is equal to the value of the switch expression, and if a `default` label is present, control is transferred to the statement list following the `default` label.
- If none of the constants specified in `case` labels in the same `switch` statement is equal to the value of the switch expression, and if no `default` label is present, control is transferred to the end point of the `switch` statement.

If the end point of the statement list of a switch section is reachable, a compile-time error occurs. This is known as the "no fall through" rule. The example

```
switch (i) {
case 0:
    CaseZero();
    break;
case 1:
    CaseOne();
    break;
default:
    CaseOthers();
    break;
}
```

is valid because no switch section has a reachable end point. Unlike C and C++, execution of a switch section is not permitted to "fall through" to the next switch section, and the example

```
switch (i) {
case 0:
    CaseZero();
case 1:
    CaseZeroOrOne();
default:
    CaseAny();
}
```

results in a compile-time error. When execution of a switch section is to be followed by execution of another switch section, an explicit `goto case` or `goto default` statement must be used:

```
switch (i) {
case 0:
    CaseZero();
    goto case 1;
case 1:
    CaseZeroOrOne();
    goto default;
default:
    CaseAny();
    break;
}
```


www.it-ebooks.info

Multiple labels are permitted in a switch section. The example

```
switch (i) {
case 0:
    CaseZero();
    break;
case 1:
    CaseOne();
    break;
case 2:
default:
    CaseTwo();
    break;
}
```

is valid. The example does not violate the "no fall through" rule because the labels `case 2:` and `default:` are part of the same switch section.

The "no fall through" rule prevents a common class of bugs that occur in C and C++ when `break` statements are accidentally omitted. In addition, because of this rule, the switch sections of a `switch` statement can be arbitrarily rearranged without affecting the behavior of the statement. For example, the sections of the `switch` statement above can be reversed without affecting the behavior of the statement:

```
switch (i) {
default:
    CaseAny();
    break;
case 1:
    CaseZeroOrOne();
    goto default;
case 0:
    CaseZero();
    goto case 1;
}
```

The statement list of a switch section typically ends in a `break`, `goto case`, or `goto default` statement, but any construct that renders the end point of the statement list unreachable is permitted. For example, a `while` statement controlled by the boolean expression `true` is known to never reach its end point. Likewise, a `throw` or `return` statement always transfers control elsewhere and never reaches its end point. Thus the following example is valid:

```
switch (i) {
case 0:
    while (true) F();
case 1:
    throw new ArgumentException();
case 2:
    return;
}
```


www.it-ebooks.info

JON SKEET I wonder whether it might have been wise for the C# designers to have redesigned `switch/case` from scratch. The scoping rules for variables introduced in cases are somewhat surprising, and the `break` statement feels wrong, too. I believe developers think of cases as blocks—so why not enforce that?

```
case 0:
{
    // Code for case 0 goes here, with no need for a break
}
```

Likewise, it would probably have been more readable to allow a comma-separated list of expressions for multiple cases to match a single block, rather than repeating whole `case` labels.

The governing type of a `switch` statement may be the type `string`. For example:

```
void DoCommand(string command) {
    switch (command.ToLower()) {
    case "run":
        DoRun();
        break;
    case "save":
        DoSave();
        break;
    case "quit":
        DoQuit();
        break;
    default:
        InvalidCommand(command);
        break;
    }
}
```

Like the string equality operators (§7.10.7), the `switch` statement is case sensitive and will execute a given switch section only if the switch expression string exactly matches a `case` label constant.

When the governing type of a `switch` statement is `string`, the value `null` is permitted as a `case` label constant.

The *statement-list*s of a *switch-block* may contain declaration statements (§8.5). The scope of a local variable or constant declared in a switch block is the switch block.

BILL WAGNER This idea suggests that implicit braces surround each switch block.

www.it-ebooks.info

Within a switch block, the meaning of a name used in an expression context must always be the same (§7.6.2.1).

The statement list of a given switch section is reachable if the `switch` statement is reachable and at least one of the following is true:

- The switch expression is a nonconstant value.
- The switch expression is a constant value that matches a `case` label in the switch section.
- The switch expression is a constant value that doesn't match any `case` label, and the switch section contains the `default` label.
- A `switch` label of the switch section is referenced by a reachable `goto case` or `goto default` statement.

> **VLADIMIR RESHETNIKOV** This rule is not applied when the `goto case` or `goto default` statement is inside a `try` or `catch` block of a `try` statement that includes a `finally` block, and the `switch` label is outside the `try` statement, and the end point of the `finally` block is unreachable.

The end point of a `switch` statement is reachable if at least one of the following is true:

- The `switch` statement contains a reachable `break` statement that exits the `switch` statement.

> **VLADIMIR RESHETNIKOV** This rule is not applied when the `break` statement is inside a `try` or `catch` block of a `try` statement that includes a `finally` block, and the target of the `break` statement is outside the `try` statement, and the end point of the `finally` block is unreachable.

- The `switch` statement is reachable, the switch expression is a nonconstant value, and no `default` label is present.
- The `switch` statement is reachable, the switch expression is a constant value that doesn't match any `case` label, and no `default` label is present.

www.it-ebooks.info

8.8 Iteration Statements

Iteration statements repeatedly execute an embedded statement.

iteration-statement:
while-statement
do-statement
for-statement
foreach-statement

8.8.1 The while Statement

The `while` statement conditionally executes an embedded statement zero or more times.

while-statement:
`while` (*boolean-expression*) *embedded-statement*

A `while` statement is executed as follows:

- The *boolean-expression* (§7.20) is evaluated.
- If the boolean expression yields `true`, control is transferred to the embedded statement. When and if control reaches the end point of the embedded statement (possibly from execution of a `continue` statement), control is transferred to the beginning of the `while` statement.
- If the boolean expression yields `false`, control is transferred to the end point of the `while` statement.

Within the embedded statement of a `while` statement, a `break` statement (§8.9.1) may be used to transfer control to the end point of the `while` statement (thus ending iteration of the embedded statement), and a `continue` statement (§8.9.2) may be used to transfer control to the end point of the embedded statement (thus performing another iteration of the `while` statement).

The embedded statement of a `while` statement is reachable if the `while` statement is reachable and the boolean expression does not have the constant value `false`.

The end point of a `while` statement is reachable if at least one of the following is true:

- The `while` statement contains a reachable `break` statement that exits the `while` statement.

www.it-ebooks.info

VLADIMIR RESHETNIKOV This rule is not applied when the `break` statement is inside a `try` or `catch` block of a `try` statement that includes a `finally` block, and the target of the `break` statement is outside the `try` statement, and the end point of the `finally` block is unreachable.

- The `while` statement is reachable and the boolean expression does not have the constant value `true`.

8.8.2 The do Statement

The `do` statement conditionally executes an embedded statement one or more times.

do-statement:
`do` *embedded-statement* `while` `(` *boolean-expression* `)` `;`

A `do` statement is executed as follows:

- Control is transferred to the embedded statement.
- When and if control reaches the end point of the embedded statement (possibly from execution of a `continue` statement), the *boolean-expression* (§7.20) is evaluated. If the boolean expression yields `true`, control is transferred to the beginning of the `do` statement. Otherwise, control is transferred to the end point of the `do` statement.

Within the embedded statement of a `do` statement, a `break` statement (§8.9.1) may be used to transfer control to the end point of the `do` statement (thus ending iteration of the embedded statement), and a `continue` statement (§8.9.2) may be used to transfer control to the end point of the embedded statement.

The embedded statement of a `do` statement is reachable if the `do` statement is reachable.

The end point of a `do` statement is reachable if at least one of the following is true:

- The `do` statement contains a reachable `break` statement that exits the `do` statement.

VLADIMIR RESHETNIKOV This rule is not applied when the `break` statement is inside a `try` or `catch` block of a `try` statement that includes a `finally` block, and the target of the `break` statement is outside the `try` statement, and the end point of the `finally` block is unreachable.

- The end point of the embedded statement is reachable and the boolean expression does not have the constant value `true`.

www.it-ebooks.info

8.8.3 The for Statement

The `for` statement evaluates a sequence of initialization expressions and then, while a condition is true, repeatedly executes an embedded statement and evaluates a sequence of iteration expressions.

for-statement:
    `for` `(` *for-initializer*$_{opt}$ `;` *for-condition*$_{opt}$ `;` *for-iterator*$_{opt}$ `)` *embedded-statement*

for-initializer:
    local-variable-declaration
    statement-expression-list

for-condition:
    boolean-expression

for-iterator:
    statement-expression-list

statement-expression-list:
    statement-expression
    statement-expression-list `,` *statement-expression*

The *for-initializer*, if present, consists of either a *local-variable-declaration* (§8.5.1) or a list of *statement-expression*s (§8.6) separated by commas. The scope of a local variable declared by a *for-initializer* starts at the *local-variable-declarator* for the variable and extends to the end of the embedded statement. The scope includes the *for-condition* and the *for-iterator*.

The *for-condition*, if present, must be a *boolean-expression* (§7.20).

The *for-iterator*, if present, consists of a list of *statement-expression*s (§8.6) separated by commas.

A `for` statement is executed as follows:

- If a *for-initializer* is present, the variable initializers or statement expressions are executed in the order they are written. This step is performed only once.
- If a *for-condition* is present, it is evaluated.
- If the *for-condition* is not present or if the evaluation yields `true`, control is transferred to the embedded statement. When and if control reaches the end point of the embedded statement (possibly from execution of a `continue` statement), the expressions of the *for-iterator*, if any, are evaluated in sequence, and then another iteration is performed, starting with evaluation of the *for-condition* in the step above.
- If the *for-condition* is present and the evaluation yields `false`, control is transferred to the end point of the `for` statement.

www.it-ebooks.info

Within the embedded statement of a `for` statement, a `break` statement (§8.9.1) may be used to transfer control to the end point of the `for` statement (thus ending iteration of the embedded statement), and a `continue` statement (§8.9.2) may be used to transfer control to the end point of the embedded statement (thus executing the *for-iterator* and performing another iteration of the `for` statement, starting with the *for-condition*).

The embedded statement of a `for` statement is reachable if one of the following is true:

- The `for` statement is reachable and no *for-condition* is present.
- The `for` statement is reachable and a *for-condition* is present and does not have the constant value `false`.

The end point of a `for` statement is reachable if at least one of the following is true:

- The `for` statement contains a reachable `break` statement that exits the `for` statement.

> **VLADIMIR RESHETNIKOV** This rule is not applied when the `break` statement is inside a `try` or `catch` block of a `try` statement that includes a `finally` block, and the target of the `break` statement is outside the `try` statement, and the end point of the `finally` block is unreachable.

- The `for` statement is reachable and a *for-condition* is present and does not have the constant value `true`.

8.8.4 The foreach Statement

The `foreach` statement enumerates the elements of a collection, executing an embedded statement for each element of the collection.

foreach-statement:
`foreach` `(` *local-variable-type* *identifier* `in` *expression* `)` *embedded-statement*

The *type* and *identifier* of a `foreach` statement declare the ***iteration variable*** of the statement. If the `var` identifier is given as the *local-variable-type*, and no type named `var` is in scope, the iteration variable is said to be an ***implicitly typed iteration variable***, and its type is taken to be the element type of the `foreach` statement, as specified below. The iteration variable corresponds to a read-only local variable with a scope that extends over the embedded statement. During execution of a `foreach` statement, the iteration variable represents the collection element for which an iteration is currently being performed. A compile-time error occurs if the embedded statement attempts to modify the iteration variable (via assignment or the `++` and `--` operators) or pass the iteration variable as a `ref` or `out` parameter.

www.it-ebooks.info

CHRIS SELLS For readability, `foreach` statements should be preferred over `for` statements.

JON SKEET The fact that there is *one* iteration variable (which is read-only and yet magically changes its value on each iteration) causes one of the most common problems with captured variables. The code below looks like it would print "a", "b", "c", "d" but it actually prints "d" four times.

```
List<Action> actions = new List<Action>();
foreach (string value in new[] { "a", "b", "c", "d" })
{
    actions.Add(() => Console.WriteLine(value));
}
foreach (Action action in actions)
{
    action();
}
```

The solution is usually to introduce an extra variable inside the body of the `foreach` statement that takes a copy of the iteration variable's current value. That way each delegate will capture a different variable. In this case, the first loop would become

```
foreach (string value in new[] { "a", "b", "c", "d" })
{
    String copy = value;
    actions.Add(() => Console.WriteLine(copy));
}
```

PETER SESTOFT The `foreach` statement might have been defined differently to avoid the variable capture problem discussed in Jon Skeet's comment: Simply move the declaration `V v` inside the `while` loop in the `try-while-finally` expansion shown later in this section. Indeed, in the original C# language specification, the declaration appeared inside the `while` loop, but during standardization of C# 2.0 it turned out that declaring `v` outside the loop was closer to the truth, at least in Microsoft's implementation, and the standard was changed accordingly.

The compile-time processing of a `foreach` statement first determines the ***collection type***, ***enumerator type***, and ***element type*** of the expression. This determination proceeds as follows:

www.it-ebooks.info

- If the type `X` of *expression* is an array type, then there is an implicit reference conversion from `X` to the `System.Collections.IEnumerable` interface (since `System.Array` implements this interface). The ***collection type*** is the `System.Collections.IEnumerable` interface, the ***enumerator type*** is the `System.Collections.IEnumerator` interface, and the ***element type*** is the element type of the array type `X`.
- If the type `X` of *expression* is `dynamic`, then there is an implicit conversion from *expression* to the `System.Collections.IEnumerable` interface (§6.1.8). The ***collection type*** is the `System.Collections.IEnumerable` interface and the ***enumerator type*** is the `System.Collections.IEnumerator` interface. If the `var` identifier is given as the *local-variable-type*, then the ***element type*** is `dynamic`; otherwise, it is `object`.
- Otherwise, determine whether the type `X` has an appropriate `GetEnumerator` method:
  - Perform member lookup on the type `X` with identifier `GetEnumerator` and no type arguments. If the member lookup does not produce a match, or if it produces an ambiguity or a match that is not a method group, check for an enumerable interface as described below. It is recommended that a warning be issued if member lookup produces anything except a method group or no match.

ERIC LIPPERT This "pattern"-based approach was specified so that back in the days before the generic `IEnumerable<T>` was available, collection authors could provide stronger type annotations on their enumerator objects.

Implementations of nongeneric `IEnumerable` always end up boxing every member of a collection of integers because the return type of the `Current` property is `object`. A provider of a collection of integers could provide non-interface-based `GetEnumerator`, `MoveNext`, and `Current` implementations such that `Current` returns an unboxed integer.

Of course, in a world with generic `IEnumerable<T>`, all of this effort becomes unnecessary. The vast majority of iterated collections will implement this interface.

  - Perform overload resolution using the resulting method group and an empty argument list. If overload resolution results in no applicable methods, results in an ambiguity, or results in a single best method but that method is either static or not `public`, check for an enumerable interface as described below. It is recommended that a warning be issued if overload resolution produces anything except an unambiguous `public` instance method or no applicable methods.
  - If the return type `E` of the `GetEnumerator` method is not a class, struct, or interface type, an error is produced and no further steps are taken.

www.it-ebooks.info

  - Member lookup is performed on `E` with the identifier `Current` and no type arguments. If the member lookup produces no match, the result is an error, or the result is anything except a `public` instance property that permits reading, an error is produced and no further steps are taken.
  - Member lookup is performed on `E` with the identifier `MoveNext` and no type arguments. If the member lookup produces no match, the result is an error, or the result is anything except a method group, an error is produced and no further steps are taken.
  - Overload resolution is performed on the method group with an empty argument list. If overload resolution results in no applicable methods, results in an ambiguity, or results in a single best method but that method is either static or not `public`, or its return type is not `bool`, an error is produced and no further steps are taken.
  - The ***collection type*** is `X`, the ***enumerator type*** is `E`, and the ***element type*** is the type of the `Current` property.
- Otherwise, check for an enumerable interface:
  - If there is exactly one type `T` such that there is an implicit conversion from `X` to the interface `System.Collections.Generic.IEnumerable<T>`, then the ***collection type*** is this interface, the ***enumerator type*** is the interface `System.Collections.Generic.IEnumerator<T>`, and the ***element type*** is `T`.
  - Otherwise, if there is more than one such type `T`, then an error is produced and no further steps are taken.
  - Otherwise, if there is an implicit conversion from `X` to the `System.Collections.IEnumerable` interface, then the ***collection type*** is this interface, the ***enumerator type*** is the interface `System.Collections.IEnumerator`, and the ***element type*** is `object`.
  - Otherwise, an error is produced and no further steps are taken.

PETER SESTOFT Sadly, the compile-time processing of the `foreach` statement means that it is rather dynamically typed. For instance, for an arbitrary non-sealed class `C` and interface `I`, the following is type correct and causes no compiler warnings or errors:

```
C[] xs = ...;
foreach (I x in xs)
   ...
```


www.it-ebooks.info

In fact, just recently I fell into this trap when removing a seemingly irrelevant interface `I` from the interface list of a class `C`. The project went through the build stage without errors, but at runtime the application crashed with an `InvalidCastException` because I had overlooked several `foreach` statements. Actually, this behavior should be expected, because there might be some subclass of `C` that implements interface `I`. What is perhaps more surprising is that `I` and `C` can be swapped in the preceding code with the same result.

The above steps, if successful, unambiguously produce a collection type `C`, enumerator type `E`, and element type `T`. A `foreach` statement of the form

```
foreach (V v in x) embedded-statement
```

is then expanded to:

```
{
    E e = ((C)(x)).GetEnumerator();
    try {
        V v;
        while (e.MoveNext()) {
            v = (V)(T)e.Current;
            embedded-statement
        }
    }
    finally {
        ... // Dispose of e
    }
}
```

The variable `e` is not visible to or accessible to the expression `x`, the embedded statement, or any other source code of the program. The variable `v` is read-only in the embedded statement. If there is not an explicit conversion (§6.2) from `T` (the element type) to `V` (the *local-variable-type* in the `foreach` statement), an error is produced and no further steps are taken. If `x` has the value `null`, a `System.NullReferenceException` is thrown at runtime.

MAREK SAFAR The `foreach` statement is a classic example of language evolution. C# 1.0 did not have the generic `IEnumerable<T>` and explicit conversion had to be used from element type to local variable. C# 2.0 introduced a generic version of the `foreach` statement with an element of a generic type. C# 3.0 brought this structure back to where it should have been at the beginning by introducing ***implicitly typed iteration variable,*** which changes explicit conversion to be implicit and avoids any `InvalidCastException` during execution.

www.it-ebooks.info

An implementation is permitted to implement a given `foreach` statement differently—for example, for performance reasons—as long as the behavior is consistent with the above expansion.

The body of the `finally` block is constructed according to the following steps:

- If there is an implicit conversion from `E` to the `System.IDisposable` interface, then
  - If `E` is a non-nullable value type, then the `finally` clause is expanded to the semantic equivalent of

```
finally {
    ((System.IDisposable)e).Dispose();
}
```

  - Otherwise, the `finally` clause is expanded to the semantic equivalent of

```
finally {
    if (e != null) ((System.IDisposable)e).Dispose();
}
```

    except that if `E` is a value type, or a type parameter instantiated to a value type, then the cast of `e` to `System.IDisposable` will not cause boxing to occur.
- Otherwise, if `E` is a sealed type, then the `finally` clause is expanded to an empty block:

```
finally {
}
```

- Otherwise, the `finally` clause is expanded to

```
finally {
    System.IDisposable d = e as System.IDisposable;
    if (d != null) d.Dispose();
}
```

  The local variable `d` is not visible to or accessible to any user code. In particular, it does not conflict with any other variable whose scope includes the `finally` block.

> **JON SKEET** The fact that the `foreach` statement disposes of its iterator makes the iterator blocks in C# 2.0 vastly more useful than they otherwise would be: It is entirely reasonable to acquire a resource to iterate over, and then dispose of that resource when either the iterator has been exhausted or the caller breaks out of the loop for some reason.

www.it-ebooks.info

The order in which `foreach` traverses the elements of an array is as follows: For single-dimensional arrays, elements are traversed in increasing index order, starting with index `0` and ending with index `Length - 1`. For multi-dimensional arrays, elements are traversed such that the indices of the rightmost dimension are increased first, then the next left dimension, and so on to the left.

The following example prints out each value in a two-dimensional array, in element order:

```
using System;

class Test
{
    static void Main()
    {
        double[,] values = {
              {1.2, 2.3, 3.4, 4.5},
              {5.6, 6.7, 7.8, 8.9}
        };

        foreach (double elementValue in values)
              Console.Write("{0} ", elementValue);

              Console.WriteLine();
    }
}
```

The output produced is as follows:

```
1.2 2.3 3.4 4.5 5.6 6.7 7.8 8.9
```

In the example

```
int[] numbers = { 1, 3, 5, 7, 9 };
foreach (var n in numbers) Console.WriteLine(n);
```

the type of `n` is inferred to be `int`, the element type of `numbers`.

8.9 Jump Statements

Jump statements unconditionally transfer control.

jump-statement:
break-statement
continue-statement
goto-statement
return-statement
throw-statement

www.it-ebooks.info

The location to which a jump statement transfers control is called the ***target*** of the jump statement.

When a jump statement occurs within a block, and the target of that jump statement is outside that block, the jump statement is said to ***exit*** the block. While a jump statement may transfer control out of a block, it can never transfer control into a block.

Execution of jump statements is complicated by the presence of intervening `try` statements. In the absence of such `try` statements, a jump statement unconditionally transfers control from the jump statement to its target. In the presence of such intervening `try` statements, execution is more complex. If the jump statement exits one or more `try` blocks with associated `finally` blocks, control is initially transferred to the `finally` block of the innermost `try` statement. When and if control reaches the end point of a `finally` block, control is transferred to the `finally` block of the next enclosing `try` statement. This process is repeated until the `finally` blocks of all intervening `try` statements have been executed.

In the example

```
using System;

class Test
{
    static void Main()
    {
        while (true)
        {
            try
            {
                try
                {
                    Console.WriteLine("Before break");
                    break;
                }
                finally
                {
                    Console.WriteLine("Innermost finally block");
                }
            }
            finally
            {
                Console.WriteLine("Outermost finally block");
            }
        }
        Console.WriteLine("After break");
    }
}
```

the `finally` blocks associated with two `try` statements are executed before control is transferred to the target of the jump statement.

The output produced is as follows:

www.it-ebooks.info

```
Before break
Innermost finally block
Outermost finally block
After break
```

8.9.1 The break Statement

The `break` statement exits the nearest enclosing `switch`, `while`, `do`, `for`, or `foreach` statement.

break-statement:
    `break ;`

The target of a `break` statement is the end point of the nearest enclosing `switch`, `while`, `do`, `for`, or `foreach` statement. If a `break` statement is not enclosed by a `switch`, `while`, `do`, `for`, or `foreach` statement, a compile-time error occurs.

When multiple `switch`, `while`, `do`, `for`, or `foreach` statements are nested within each other, a `break` statement applies only to the innermost statement. To transfer control across multiple nesting levels, a `goto` statement (§8.9.3) must be used.

A `break` statement cannot exit a `finally` block (§8.10). When a `break` statement occurs within a `finally` block, the target of the `break` statement must be within the same `finally` block; otherwise, a compile-time error occurs.

A `break` statement is executed as follows:

- If the `break` statement exits one or more `try` blocks with associated `finally` blocks, control is initially transferred to the `finally` block of the innermost `try` statement. When and if control reaches the end point of a `finally` block, control is transferred to the `finally` block of the next enclosing `try` statement. This process is repeated until the `finally` blocks of all intervening `try` statements have been executed.
- Control is transferred to the target of the `break` statement.

Because a `break` statement unconditionally transfers control elsewhere, the end point of a `break` statement is never reachable.

JESSE LIBERTY Whenever you see unconditional transfer or (especially) exit from within the flow of a method, treat it as a volatile explosive (that is, run screaming from the room).

While a case can be made for `break`, `continue`, and `goto` statements, it is rare that you can't rewrite or refactor them out of existence. The resulting code is very likely to be easier to read, understand, and maintain.

www.it-ebooks.info

8.9.2 The continue Statement

The `continue` statement starts a new iteration of the nearest enclosing `while`, `do`, `for`, or `foreach` statement.

continue-statement:
`continue ;`

The target of a `continue` statement is the end point of the embedded statement of the nearest enclosing `while`, `do`, `for`, or `foreach` statement. If a `continue` statement is not enclosed by a `while`, `do`, `for`, or `foreach` statement, a compile-time error occurs.

When multiple `while`, `do`, `for`, or `foreach` statements are nested within each other, a `continue` statement applies only to the innermost statement. To transfer control across multiple nesting levels, a `goto` statement (§8.9.3) must be used.

A `continue` statement cannot exit a `finally` block (§8.10). When a `continue` statement occurs within a `finally` block, the target of the `continue` statement must be within the same `finally` block; otherwise, a compile-time error occurs.

A `continue` statement is executed as follows:

- If the `continue` statement exits one or more `try` blocks with associated `finally` blocks, control is initially transferred to the `finally` block of the innermost `try` statement. When and if control reaches the end point of a `finally` block, control is transferred to the `finally` block of the next enclosing `try` statement. This process is repeated until the `finally` blocks of all intervening `try` statements have been executed.
- Control is transferred to the target of the `continue` statement.

Because a `continue` statement unconditionally transfers control elsewhere, the end point of a `continue` statement is never reachable.

PETER SESTOFT Some languages—notably the Java programming language—have generalized versions of the `break` and `continue` statements, where a target label specifies which of multiple enclosing loops should be exited or resumed. The same effect can be obtained in C# using the `goto` statement. Moreover, although the generalized `break` and `continue` statements are disciplined versions of `goto`, they are almost as difficult to understand as `goto` statements and, in my opinion, detrimental to readability even in short programs. See examples 78 and 79 in my book *Java Precisely*, second edition.

www.it-ebooks.info

8.9.3 The goto Statement

The `goto` statement transfers control to a statement that is marked by a label.

goto-statement:
    `goto` *identifier* `;`
    `goto` `case` *constant-expression* `;`
    `goto` `default` `;`

The target of a `goto` *identifier* statement is the labeled statement with the given label. If a label with the given name does not exist in the current function member, or if the `goto` statement is not within the scope of the label, a compile-time error occurs. This rule permits the use of a `goto` statement to transfer control *out of* a nested scope, but not *into* a nested scope. In the example

```
using System;

class Test
{
    static void Main(string[] args) {
        string[,] table = {
            {"Red", "Blue", "Green"},
            {"Monday", "Wednesday", "Friday"}
        };

        foreach (string str in args) {
            int row, colm;
            for (row = 0; row <= 1; ++row)
                for (colm = 0; colm <= 2; ++colm)
                    if (str == table[row,colm])
                        goto done;

            Console.WriteLine("{0} not found", str);
            continue;
    done:
            Console.WriteLine("Found {0} at [{1}][{2}]", str, row, colm);
        }
    }
}
```

a `goto` statement is used to transfer control out of a nested scope.

The target of a `goto case` statement is the statement list in the immediately enclosing `switch` statement (§8.7.2), which contains a `case` label with the given constant value. If the `goto case` statement is not enclosed by a `switch` statement, if the *constant-expression* is not implicitly convertible (§6.1) to the governing type of the nearest enclosing `switch` statement, or if the nearest enclosing `switch` statement does not contain a `case` label with the given constant value, a compile-time error occurs.

The target of a `goto default` statement is the statement list in the immediately enclosing `switch` statement (§8.7.2), which contains a `default` label. If the `goto default` statement is

www.it-ebooks.info

not enclosed by a `switch` statement, or if the nearest enclosing `switch` statement does not contain a `default` label, a compile-time error occurs.

A `goto` statement cannot exit a `finally` block (§8.10). When a `goto` statement occurs within a `finally` block, the target of the `goto` statement must be within the same `finally` block; otherwise, a compile-time error occurs.

> **BILL WAGNER** These rules make `goto` a little less than pure evil in C#, but I've yet to see a good use for it.

A `goto` statement is executed as follows:

- If the `goto` statement exits one or more `try` blocks with associated `finally` blocks, control is initially transferred to the `finally` block of the innermost `try` statement. When and if control reaches the end point of a `finally` block, control is transferred to the `finally` block of the next enclosing `try` statement. This process is repeated until the `finally` blocks of all intervening `try` statements have been executed.
- Control is transferred to the target of the `goto` statement.

Because a `goto` statement unconditionally transfers control elsewhere, the end point of a `goto` statement is never reachable.

> **CHRIS SELLS** Please don't use labels or `goto` statements. I've never read any code that wasn't more readable without them.

> **CHRISTIAN NAGEL** Although `goto` statements make sense within `switch` statements as explicit fall-through options, you shouldn't use them in other scenarios.

> **PETER SESTOFT** Don Knuth's 1974 paper "Structured Programming with go to Statements" throws a lot of light on alternatives to the `goto` statement and identifies sensible uses of the `goto` statement. In the end, Knuth does conclude that "we should, indeed, abolish go to . . . , at least as an experiment in training people to formulate their abstractions more carefully." Just the acknowledgments list of that paper presents a "who's who" of programming language pioneers.
>
> The best reason for including `goto` in C# probably is that you may want to *generate* code that uses `goto`—for instance, parsers, lexers, automata, statecharts, virtual machines, and similar programs.

www.it-ebooks.info

8.9.4 The return Statement

The `return` statement returns control to the caller of the function member in which the `return` statement appears.

return-statement:
`return` *expression*$_{opt}$ `;`

A `return` statement with no expression can be used only in a function member that does not compute a value—that is, a method with the return type `void`, the `set` accessor of a property or indexer, the `add` and `remove` accessors of an event, an instance constructor, a static constructor, or a destructor.

A `return` statement with an expression can only be used in a function member that computes a value—that is, a method with a non-void return type, the `get` accessor of a property or indexer, or a user-defined operator. An implicit conversion (§6.1) must exist from the type of the expression to the return type of the containing function member.

> **VLADIMIR RESHETNIKOV** If the `return` statement is within an anonymous function, the rules from §6.5 are applied instead.

It is a compile-time error for a `return` statement to appear in a `finally` block (§8.10).

A `return` statement is executed as follows:

- If the `return` statement specifies an expression, the expression is evaluated and the resulting value is converted to the return type of the containing function member by an implicit conversion. The result of the conversion becomes the value returned to the caller.
- If the `return` statement is enclosed by one or more `try` blocks with associated `finally` blocks, control is initially transferred to the `finally` block of the innermost `try` statement. When and if control reaches the end point of a `finally` block, control is transferred to the `finally` block of the next enclosing `try` statement. This process is repeated until the `finally` blocks of all enclosing `try` statements have been executed.
- Control is returned to the caller of the containing function member.

Because a `return` statement unconditionally transfers control elsewhere, the end point of a `return` statement is never reachable.

www.it-ebooks.info

8.9.5 The throw Statement

The `throw` statement throws an exception.

throw-statement:
`throw` *expression*$_{opt}$ `;`

A `throw` statement with an expression throws the value produced by evaluating the expression. The expression must denote a value of the class type `System.Exception`, of a class type that derives from `System.Exception` or of a type parameter type that has `System.Exception` (or a subclass thereof) as its effective base class. If evaluation of the expression produces `null`, a `System.NullReferenceException` is thrown instead.

PETER SESTOFT For this reason, an exception thrown in C# is never null and, therefore, the `try-catch` matching of an exception (on its class) described in §8.10 makes sense. In fact, the .NET/CLI intermediate language instruction called `throw` behaves like the C# `throw` statement in this respect. Thus, even if the exception was thrown by code written in another .NET/CLI language, it would be non-null when handled by the `try-catch` statement.

A `throw` statement with no expression can be used only in a `catch` block, in which case that statement rethrows the exception that is currently being handled by that `catch` block.

VLADIMIR RESHETNIKOV A `throw` statement with no *expression* is not allowed in a `finally` block or anonymous function that is nested inside the nearest enclosing `catch` block:

```
delegate void F();
class Program
{
    static void Main()
    {
        try
        {
        }
        catch
        {
            F f = () => { throw; }; // Error CS0156
            try
            {
            }
            finally
            {
                throw;              // Error CS0724
            }
        }
    }
}
```


www.it-ebooks.info

Because a `throw` statement unconditionally transfers control elsewhere, the end point of a `throw` statement is never reachable.

When an exception is thrown, control is transferred to the first `catch` clause in an enclosing `try` statement that can handle the exception. The process that takes place from the point of the exception being thrown to the point of transferring control to a suitable exception handler is known as ***exception propagation***. Propagation of an exception consists of repeatedly evaluating the following steps until a `catch` clause that matches the exception is found. In this description, the ***throw point*** is initially the location at which the exception is thrown.

- In the current function member, each `try` statement that encloses the throw point is examined. For each statement `S`, starting with the innermost `try` statement and ending with the outermost `try` statement, the following steps are evaluated:
  - If the `try` block of `S` encloses the throw point and if `S` has one or more `catch` clauses, the `catch` clauses are examined in order of appearance to locate a suitable handler for the exception. The first `catch` clause that specifies the exception type or a base type of the exception type is considered a match. A general `catch` clause (§8.10) is considered a match for any exception type. If a matching `catch` clause is located, the exception propagation is completed by transferring control to the block of that `catch` clause.
  - Otherwise, if the `try` block or a `catch` block of `S` encloses the throw point and if `S` has a `finally` block, control is transferred to the `finally` block. If the `finally` block throws another exception, processing of the current exception is terminated. Otherwise, when control reaches the end point of the `finally` block, processing of the current exception is continued.
- If an exception handler was not located in the current function member invocation, the function member invocation is terminated. The steps above are then repeated for the caller of the function member with a throw point corresponding to the statement from which the function member was invoked.
- If the exception processing terminates all function member invocations in the current thread, indicating that the thread has no handler for the exception, then the thread is itself terminated. The impact of such termination is implementation-defined.

BILL WAGNER This process implies that you should do some basic cleanup in the topmost methods for all your threads. Otherwise, the behavior of your application will be undefined in the face of exceptions that will cause threads to be terminated.

www.it-ebooks.info

8.10 The try Statement

The `try` statement provides a mechanism for catching exceptions that occur during execution of a block. Furthermore, the `try` statement provides the ability to specify a block of code that is always executed when control leaves the `try` statement.

try-statement:
    `try` *block* *catch-clauses*
    `try` *block* *finally-clause*
    `try` *block* *catch-clauses* *finally-clause*

catch-clauses:
    specific-catch-clauses *general-catch-clause*$_{opt}$
    specific-catch-clauses$_{opt}$ *general-catch-clause*

specific-catch-clauses:
    specific-catch-clause
    specific-catch-clauses *specific-catch-clause*

specific-catch-clause:
    `catch` `(` *class-type* *identifier*$_{opt}$ `)` *block*

general-catch-clause:
    `catch` *block*

finally-clause:
    `finally` *block*

There are three possible forms of `try` statements:

- A `try` block followed by one or more `catch` blocks.
- A `try` block followed by a `finally` block.
- A `try` block followed by one or more `catch` blocks followed by a `finally` block.

When a `catch` clause specifies a *class-type*, the type must be `System.Exception`, a type that derives from `System.Exception`, or a type parameter type that has `System.Exception` (or a subclass thereof) as its effective base class.

When a `catch` clause specifies both a *class-type* and an *identifier*, an ***exception variable*** of the given name and type is declared. The exception variable corresponds to a local variable

www.it-ebooks.info

with a scope that extends over the `catch` block. During execution of the `catch` block, the exception variable represents the exception currently being handled. For purposes of definite assignment checking, the exception variable is considered definitely assigned in its entire scope.

Unless a `catch` clause includes an exception variable name, it is impossible to access the exception object in the `catch` block.

A `catch` clause that specifies neither an exception type nor an exception variable name is called a general `catch` clause. A `try` statement can have only one general `catch` clause, and if one is present it must be the last `catch` clause.

Some programming languages may support exceptions that are not representable as an object derived from `System.Exception`, although such exceptions could never be generated by C# code. A general `catch` clause may be used to catch such exceptions. Thus a general `catch` clause is semantically different from one that specifies the type `System.Exception`, in that the former may also catch exceptions from other languages.

ERIC LIPPERT In the current Microsoft implementation of C# and the CLR, by default a thrown object that does not derive from `Exception` is converted into a `RuntimeWrappedException` object. As a consequence, `catch(Exception e)` catches all exceptions.

If you want to disable this behavior and use the C# 1.0 semantics, whereby non-`Exception` objects thrown by other languages are not caught in this manner, then use the following assembly attribute:

```
[assembly:System.Runtime.CompilerServices.RuntimeCompatibility(WrapNonExceptionThrows
= false)]
```

To locate a handler for an exception, `catch` clauses are examined in lexical order. A compile-time error occurs if a `catch` clause specifies a type that is the same as, or is derived from, a type that was specified in an earlier `catch` clause for the same `try` block. Without this restriction, it would be possible to write unreachable `catch` clauses.

Within a `catch` block, a `throw` statement (§8.9.5) with no expression can be used to rethrow the exception that was caught by the `catch` block. Assignments to an exception variable do not alter the exception that is rethrown.

www.it-ebooks.info

In the example

```
using System;

class Test
{
    static void F() {
        try {
            G();
        }
        catch (Exception e) {
            Console.WriteLine("Exception in F: " + e.Message);
            e = new Exception("F");
            throw;                      // Rethrow
        }
    }

    static void G() {
        throw new Exception("G");
    }

    static void Main() {
        try {
            F();
        }
        catch (Exception e) {
            Console.WriteLine("Exception in Main: " + e.Message);
        }
    }
}
```

the method `F` catches an exception, writes some diagnostic information to the console, alters the exception variable, and rethrows the exception. The exception that is rethrown is the original exception, so the output produced is

```
Exception in F: G
Exception in Main: G
```

If the first `catch` block had thrown `e` instead of rethrowing the current exception, the output produced would be as follows:

```
Exception in F: G
Exception in Main: F
```

It is a compile-time error for a `break`, `continue`, or `goto` statement to transfer control out of a `finally` block. When a `break`, `continue`, or `goto` statement occurs in a `finally` block, the target of the statement must be within the same `finally` block; otherwise, a compile-time error occurs.

www.it-ebooks.info

It is a compile-time error for a `return` statement to occur in a `finally` block.

A `try` statement is executed as follows:

- Control is transferred to the `try` block.
- When and if control reaches the end point of the `try` block:
  - If the `try` statement has a `finally` block, the `finally` block is executed.
  - Control is transferred to the end point of the `try` statement.
- If an exception is propagated to the `try` statement during execution of the `try` block:
  - The `catch` clauses, if any, are examined in order of appearance to locate a suitable handler for the exception. The first `catch` clause that specifies the exception type or a base type of the exception type is considered a match. A general `catch` clause is considered a match for any exception type. If a matching `catch` clause is located:
    - If the matching `catch` clause declares an exception variable, the exception object is assigned to the exception variable.
    - Control is transferred to the matching `catch` block.
    - When and if control reaches the end point of the `catch` block:
      - If the `try` statement has a `finally` block, the `finally` block is executed.
      - Control is transferred to the end point of the `try` statement.
    - If an exception is propagated to the `try` statement during execution of the `catch` block:
      - If the `try` statement has a `finally` block, the `finally` block is executed.
      - The exception is propagated to the next enclosing `try` statement.
  - If the `try` statement has no `catch` clauses or if no `catch` clause matches the exception:
    - If the `try` statement has a `finally` block, the `finally` block is executed.
    - The exception is propagated to the next enclosing `try` statement.

www.it-ebooks.info

ERIC LIPPERT If the call stack includes code protected by `try-catch` blocks written in other languages (such as Visual Basic), the runtime environment may execute an "exception filter" to see whether a given `catch` block is appropriate for the thrown exception. As a consequence, user code may execute after an exception is thrown but before the associated `finally` block is executed. If your exception-handling code depends on the global state being made consistent by a `finally` block before any other user code runs, then you should take appropriate measures to ensure that your code catches the exception before the runtime environment executes user-defined exception filters that may be on the stack.

The statements of a `finally` block are always executed when control leaves a `try` statement. This is true whether the control transfer occurs as a result of normal execution; as a result of executing a `break`, `continue`, `goto`, or `return` statement; or as a result of propagating an exception out of the `try` statement.

If an exception is thrown during execution of a `finally` block and is not caught within the same `finally` block, the exception is propagated to the next enclosing `try` statement. If another exception was in the process of being propagated, that exception is lost. The process of propagating an exception is discussed further in the description of the `throw` statement (§8.9.5).

BILL WAGNER This behavior makes it very important to write `finally` clauses defensively to avoid raising a second exception.

JON SKEET It would be nice if we *didn't* have to write `finally` clauses defensively, of course. The concept of one error being caused by another is already part of the .NET framework with the idea of an "inner exception." Even so, the idea that two exceptions are likely to be related, but neither was known to cause the other, doesn't have a common representation at the moment. Likewise, there's the generally thorny question of which exceptions a method might throw. Java tried to tackle this problem with "checked exceptions"—mostly unsuccessfully, in my view.

It sometimes feels as if we (as an industry) are getting quite good at success scenarios, but we still have a long way to go when it comes to error handling.

www.it-ebooks.info

The `try` block of a `try` statement is reachable if the `try` statement is reachable.

A `catch` block of a `try` statement is reachable if the `try` statement is reachable.

The `finally` block of a `try` statement is reachable if the `try` statement is reachable.

The end point of a `try` statement is reachable if both of the following are true:

- The end point of the `try` block is reachable or the end point of at least one `catch` block is reachable.
- If a `finally` block is present, the end point of the `finally` block is reachable.

8.11 The checked and unchecked Statements

The `checked` and `unchecked` statements are used to control the ***overflow checking context*** for integral-type arithmetic operations and conversions.

checked-statement:
`checked` *block*

unchecked-statement:
`unchecked` *block*

The `checked` statement causes all expressions in the *block* to be evaluated in a checked context. The `unchecked` statement causes all expressions in the *block* to be evaluated in an unchecked context.

The `checked` and `unchecked` statements are precisely equivalent to the `checked` and `unchecked` operators (§7.6.12), except that they operate on blocks instead of expressions.

8.12 The lock Statement

The `lock` statement obtains the mutual-exclusion lock for a given object, executes a statement, and then releases the lock.

lock-statement:
`lock` `(` *expression* `)` *embedded-statement*

The expression of a `lock` statement must denote a value of a type known to be a *reference-type*. No implicit boxing conversion (§6.1.7) is ever performed for the expression of a `lock`

www.it-ebooks.info

statement, and thus it is a compile-time error for the expression to denote a value of a *value-type*.

A `lock` statement of the form

```
lock (x) ...
```

where `x` is an expression of a *reference-type*, is precisely equivalent to

```
bool __lockWasTaken = false;
try {
     System.Threading.Monitor.Enter(x, __lockWasTaken);
    ...
}
finally {
     if (__lockWasTaken)System.Threading.Monitor.Exit(x);
}
```

except that `x` is evaluated only once.

> **BILL WAGNER** The `lock()` statement provides additional compiler checks against trying to lock a value type.

While a mutual-exclusion lock is held, code executing in the same execution thread can also obtain and release the lock. However, code executing in other threads is blocked from obtaining the lock until the lock is released.

Locking `System.Type` objects so as to synchronize access to static data is not recommended. Other code might lock on the same type, which can result in deadlock. A better approach is to synchronize access to static data by locking a private static object. For example:

```
class Cache
{
    private static readonly object synchronizationObject = new object();

    public static void Add(object x) {
        lock (Cache.synchronizationObject) {
            ...
        }
    }

    public static void Remove(object x) {
        lock (Cache.synchronizationObject) {
            ...
        }
    }
}
```


www.it-ebooks.info

JOSEPH ALBAHARI A good policy, when writing libraries for public consumption, is to make static functions thread-safe from the outset (typically, by implementing locking *within* your functions, as in this example). It's much more difficult (or even impossible) for consumers to lock *around* calling your static methods or properties, because they won't know from which other locations your functions are called.

8.13 The using Statement

The `using` statement obtains one or more resources, executes a statement, and then disposes of the resource.

ERIC LIPPERT As the specification explicitly calls out here, the point of a `using` statement is to ensure the acquisition of and timely disposal of a *resource.* Typically, this will consist of some unmanaged resource from the operating system, such as a file handle. It is polite to stop using resources as soon as possible; some other program might want to read that file when you're done with it. I recommend against the use of `using` statements to enforce program invariants. For example, one sometimes sees code like this:

```
using(new TemporarilyStopReportingErrors()) AttemptSomething();
```

Here `TemporarilyStopReportingErrors` is a type whose constructor turns off error reporting as a side effect and whose disposal method turns it back on. I consider this (unfortunately widespread) practice to be an abuse of the `using` statement; a program side effect is not a resource, and causing global side effects in constructors and disposers seems like a bad idea. I would write this code using a `try-finally` construct instead.

using-statement:
`using` (*resource-acquisition*) *embedded-statement*

resource-acquisition:
local-variable-declaration
expression

A ***resource*** is a class or struct that implements `System.IDisposable`, which includes a single parameterless method named `Dispose`. Code that is using a resource can call `Dispose` to indicate that the resource is no longer needed. If `Dispose` is not called, then automatic disposal eventually occurs as a consequence of garbage collection.

www.it-ebooks.info

JOSEPH ALBAHARI Calling `Dispose` doesn't influence garbage collection in any way: An object becomes eligible for automatic garbage collection when (and only when) no other object refers to it. Likewise, garbage collection doesn't influence disposal: The garbage collector will not call `Dispose` unless you write a finalizer (destructor) that explicitly makes this call.

The two activities most commonly performed within a `Dispose` method are releasing unmanaged resources and calling `Dispose` on other referenced or "owned" objects. It is also possible to release unmanaged resources from within a finalizer, although such an operation means waiting an indeterminate amount of time for the garbage collector to fire. This is why `IDisposable` exists.

If the form of *resource-acquisition* is *local-variable-declaration*, then the type of the *local-variable-declaration* must be either `dynamic` or a type that can be implicitly converted to `System.IDisposable`. If the form of *resource-acquisition* is *expression*, then this expression must be implicitly convertible to `System.IDisposable`.

Local variables declared in a *resource-acquisition* are read-only, and must include an initializer. A compile-time error occurs if the embedded statement attempts to modify these local variables (via assignment or the `++` and `--` operators), take the address of them, or pass them as `ref` or `out` parameters.

A `using` statement is translated into three parts: acquisition, usage, and disposal. Usage of the resource is implicitly enclosed in a `try` statement that includes a `finally` clause. This `finally` clause disposes of the resource. If a `null` resource is acquired, then no call to `Dispose` is made, and no exception is thrown. If the resource is of type `dynamic`, it is dynamically converted through an implicit dynamic conversion (§6.1.8) to `IDisposable` during acquisition to ensure that the conversion is successful before the usage and disposal take place.

A `using` statement of the form

```
using (ResourceType resource = expression) statement
```

corresponds to one of three possible expansions. When `ResourceType` is a non-nullable value type, the expansion is

```
{
    ResourceType resource = expression;
        try {
            statement;
        }
        finally {
            ((IDisposable)resource).Dispose();
        }
}
```


www.it-ebooks.info

Otherwise, when `ResourceType` is a nullable value type or a reference type other than `dynamic`, the expansion is

```
{
    ResourceType resource = expression;
    try {
        statement;
    }
    finally {
        if (resource != null) ((IDisposable)resource).Dispose();
    }
}
```

Otherwise, when `ResourceType` is `dynamic`, the expansion is

```
{
    ResourceType resource = expression;
    IDisposable d = (IDisposable)resource;
    try {
        statement;
    }
    finally {
        if (d != null) d.Dispose();
    }
}
```

In either expansion, the `resource` variable is read-only in the embedded statement, and the `d` variable is inaccessible in, and invisible to, the embedded statement.

PETER SESTOFT Thanks to the above rule, the `using` statement could also be (ab) used to declare read-only local variables, in this style:

```
using (MyClass v = ...)
using (MyStruct s = ...) {
  ...
}
```

In fact, this is the only way to declare an immutable local variable in C#, but it is ugly and strange. Moreover, it works only for types `MyClass` and `MyStruct` that implement interface `IDisposable`, and not for `int` and `string`, for example. What is perhaps puzzling from a language design point of view is that unlike a read-only field (§10.5.2 and §7.6.4) of struct type, an immutable resource variable `s` of struct type `MyStruct` is treated like a variable, not a value. Thus a method call `s.SetX()` acts on the struct stored in `s`, not on a copy of it. All in all, this behavior shows that the machinery for declaring immutable local variables and parameters in C# exists, but alas not in a usable form.

www.it-ebooks.info

An implementation is permitted to implement a given `using` statement differently—for example, for performance reasons—as long as the behavior is consistent with the above expansion.

A `using` statement of the form

```
using (expression) statement
```

has the same three possible expansions, but in this case `ResourceType` is implicitly the compile-time type of the `expression`, and the `resource` variable is inaccessible in, and invisible to, the embedded statement.

> **JON SKEET** I normally insist on braces around everything—and this preference usually extends to `using` statements, too. However, if you acquire multiple resources of different types (necessitating multiple `using` statements), you can nest them with only one set of braces:
>
> ```
> using (TextWriter output = File.CreateText("log.txt"))
> using (TextReader input = File.OpenText("log.txt")) {
>     // Copy contents from one to the other
> }
> ```
>
> This can dramatically reduce the level of indentation required, making the code much more readable.

When a *resource-acquisition* takes the form of a *local-variable-declaration*, it is possible to acquire multiple resources of a given type. A `using` statement of the form

```
using (ResourceType r1 = e1, r2 = e2, ..., rN = eN) statement
```

is precisely equivalent to a sequence of nested `using` statements:

```
using (ResourceType r1 = e1)
    using (ResourceType r2 = e2)
        ...
            using (ResourceType rN = eN)
                statement
```

The example below creates a file named `log.txt` and writes two lines of text to the file. The example then opens that same file for reading and copies the contained lines of text to the console.

```
using System;
using System.IO;
```


www.it-ebooks.info

```
class Test
{
    static void Main() {
        using (TextWriter w = File.CreateText("log.txt")) {
            w.WriteLine("This is line one");
            w.WriteLine("This is line two");
        }

        using (TextReader r = File.OpenText("log.txt")) {
            string s;
            while ((s = r.ReadLine()) != null) {
                Console.WriteLine(s);
            }

        }
    }
}
```

Since the `TextWriter` and `TextReader` classes implement the `IDisposable` interface, the example can use `using` statements to ensure that the underlying file is properly closed following the write or read operations.

CHRIS SELLS You should almost always wrap a `using` block around any resource you acquire that implements `IDisposable` (unless you're keeping that object between method invocations). Although the .NET garbage collector does a wonderful job with releasing memory resources, all other resources are yours to manage. Likewise, the compiler does a wonderful job of generating the proper disposing code for you, but only when you wrap your resource allocations in `using` blocks.

8.14 The yield Statement

BILL WAGNER The `yield return` and `yield break` statements seem to be two of the most under-appreciated statements in the C# language. They are incredibly useful whenever you are writing algorithms that work on sequences of data. Generating sequences, filtering sequences, combining sequences, and other algorithms are all built using these statements. In fact, most of LINQ to Objects is built using the `yield` statement.

If you're not already familiar with these techniques, you should spend the time to make this technique part of your everyday development practices.

www.it-ebooks.info

CHRIS SELLS I agree with Bill. I didn't appreciate just how useful `yield return` was until I saw someone write code like this:

```
IEnumerable<int> GetSomeNumbers() {
  yield return 1;
  yield return 2;
  yield return 3;
}
```

If you can get your head around that and what the compiler is doing for you to make that happen, you'll find yourself using this feature a lot more.

The `yield` statement is used in an iterator block (§8.2) to yield a value to the enumerator object (§10.14.4) or enumerable object (§10.14.5) of an iterator or to signal the end of the iteration.

yield-statement:
    `yield return` *expression* `;`
    `yield break ;`

Note that `yield` is not a reserved word; it has special meaning only when used immediately before a `return` or `break` keyword. In other contexts, `yield` can be used as an identifier.

There are several restrictions on where a `yield` statement can appear, as described in the following list:

- It is a compile-time error for a `yield` statement (of either form) to appear outside a *method-body*, *operator-body*, or *accessor-body*.
- It is a compile-time error for a `yield` statement (of either form) to appear inside an anonymous function.
- It is a compile-time error for a `yield` statement (of either form) to appear in the `finally` clause of a `try` statement.
- It is a compile-time error for a `yield return` statement to appear anywhere in a `try` statement that contains any `catch` clauses.

The following example shows some valid and invalid uses of `yield` statements.

```
delegate IEnumerable<int> D();

IEnumerator<int> GetEnumerator() {
    try {
        yield return 1;   // Okay
        yield break;      // Okay
    }
```


www.it-ebooks.info

```
    finally {
        yield return 2;   // Error: yield in finally
        yield break;      // Error: yield in finally
    }

    try {
        yield return 3;   // Error: yield return in try...catch
        yield break;      // Okay
    }
    catch {
        yield return 4;   // Error: yield return in try...catch
        yield break;      // Okay
    }

    D d = delegate {
        yield return 5;   // Error: yield in an anonymous function
    };
}

int MyMethod() {
    yield return 1;         // Error: wrong return type for an
                            // iterator block
}
```

An implicit conversion (§6.1) must exist from the type of the expression in the `yield return` statement to the yield type (§10.14.3) of the iterator.

A `yield return` statement is executed as follows:

- The expression given in the statement is evaluated, implicitly converted to the yield type, and assigned to the `Current` property of the enumerator object.
- Execution of the iterator block is suspended. If the `yield return` statement is within one or more `try` blocks, the associated `finally` blocks are *not* executed at this time.
- The `MoveNext` method of the enumerator object returns `true` to its caller, indicating that the enumerator object successfully advanced to the next item.

The next call to the enumerator object's `MoveNext` method resumes execution of the iterator block from where it was last suspended.

A `yield break` statement is executed as follows:

- If the `yield break` statement is enclosed by one or more `try` blocks with associated `finally` blocks, control is initially transferred to the `finally` block of the innermost `try` statement. When and if control reaches the end point of a `finally` block, control is transferred to the `finally` block of the next enclosing `try` statement. This process is repeated until the `finally` blocks of all enclosing `try` statements have been executed.
- Control is returned to the caller of the iterator block. This is either the `MoveNext` method or the `Dispose` method of the enumerator object.

www.it-ebooks.info

Because a `yield break` statement unconditionally transfers control elsewhere, the end point of a `yield break` statement is never reachable.

CHRIS SELLS I sometimes forget that `yield return` is not the same as `return`, in that the code after a `yield return` can be executed. For example, the code after the first `return` here can never be executed:

```
int F() {
    return 1;
    return 2; // Can never be executed
}
```

In contrast, the code after the first `yield return` here can be executed:

```
IEnumerable<int> F() {
    yield return 1;
    yield return 2; // Can be executed
}
```

This often bites me in an `if` statement:

```
IEnumerable<int> F() {
    if(...) { yield return 1; } // I mean this to be the only
                               // thing returned
    yield return 2;            // Oops!
}
```

In these cases, remembering that `yield return` is not "final" like `return` is helpful.

CHRISTIAN NAGEL Since C# 1.0, it has been easy to use iterators (the `foreach` statement). Since C# 2.0, it has been easy to create iterators (the `yield` statement).

www.it-ebooks.info

9. Namespaces

C# programs are organized using namespaces. Namespaces are used both as an "internal" organization system for a program and as an "external" organization system—a way of presenting program elements that are exposed to other programs.

Using directives (§9.4) are provided to facilitate the use of namespaces.

> **BILL WAGNER** As you read this, remember that namespaces are a logical organization: Multiple namespaces can, and often do, occur in one assembly, and a single namespace may be declared in many different assemblies.

9.1 Compilation Units

A *compilation-unit* defines the overall structure of a source file. A compilation unit consists of zero or more *using-directive*s followed by zero or more *global-attributes* followed by zero or more *namespace-member-declaration*s.

compilation-unit:
extern-alias-directives$_{opt}$ *using-directives*$_{opt}$ *global-attributes*$_{opt}$
namespace-member-declarations$_{opt}$

A C# program consists of one or more compilation units, each contained in a separate source file. When a C# program is compiled, all of the compilation units are processed together. Thus compilation units can depend on each other, possibly in a circular fashion.

> **CHRIS SELLS** This feature alone makes C# about ten times easier to work with than either C or C++.

The *using-directives* of a compilation unit affect the *global-attributes* and *namespace-member-declarations* of that compilation unit, but have no effect on other compilation units.

www.it-ebooks.info

> **JON SKEET** Most of the time I think I like this feature—but just occasionally, when I want to whip up a quick program to test just a few lines of code, I'd rather not have to import a bunch of namespaces. It would be interesting to know what the effect of having "project-level" using directives (like assembly references) would have been in an alternative reality.

> **ERIC LIPPERT** Jon's point is well taken: The "ceremony" needed to write any C# program is disporportionately large if the program is very short. Although C# is not intended to be a "scripting" language per se, there is something nice about the property that a one-line program is actually one line in, say, JScript.

The *global-attributes* (§17) of a compilation unit permit the specification of attributes for the target assembly and module. Assemblies and modules act as physical containers for types. An assembly may consist of several physically separate modules.

The *namespace-member-declarations* of each compilation unit of a program contribute members to a single declaration space called the global namespace. For example:

File `A.cs`:

```
class A {}
```

File `B.cs`:

```
class B {}
```

The two compilation units contribute to the single global namespace—in this case, declaring two classes with the fully qualified names `A` and `B`. Because the two compilation units contribute to the same declaration space, it would have been an error if each contained a declaration of a member with the same name.

9.2 Namespace Declarations

A *namespace-declaration* consists of the keyword `namespace`, followed by a namespace name and body, optionally followed by a semicolon.

namespace-declaration:
`namespace` *qualified-identifier* *namespace-body* `;`$_{opt}$

www.it-ebooks.info

qualified-identifier:
    identifier
    qualified-identifier `.` *identifier*

namespace-body:
    `{` *extern-alias-directives*$_{opt}$ *using-directives*$_{opt}$ *namespace-member-declarations*$_{opt}$ `}`

A *namespace-declaration* may occur as a top-level declaration in a *compilation-unit* or as a member declaration within another *namespace-declaration*. When a *namespace-declaration* occurs as a top-level declaration in a *compilation-unit*, the namespace becomes a member of the global namespace. When a *namespace-declaration* occurs within another *namespace-declaration*, the inner namespace becomes a member of the outer namespace. In either case, the name of a namespace must be unique within the containing namespace.

Namespaces are implicitly `public` and the declaration of a namespace cannot include any access modifiers.

Within a *namespace-body*, the optional *using-directives* import the names of other namespaces and types, allowing them to be referenced directly instead of through qualified names. The optional *namespace-member-declarations* contribute members to the declaration space of the namespace. Note that all *using-directives* must appear before any member declarations.

The *qualified-identifier* of a *namespace-declaration* may be a single identifier or a sequence of identifiers separated by "`.`" tokens. The latter form permits a program to define a nested namespace without lexically nesting several namespace declarations. For example,

```
namespace N1.N2
{
    class A { }

    class B { }
}
```

is semantically equivalent to

```
namespace N1
{
    namespace N2
    {
        class A { }

        class B { }
    }
}
```


www.it-ebooks.info

Namespaces are open-ended, and two namespace declarations with the same fully qualified name contribute to the same declaration space (§3.3). In the example

```
namespace N1.N2
{
    class A { }
}

namespace N1.N2
{
    class B { }
}
```

the two namespace declarations contribute to the same declaration space—in this case declaring two classes with the fully qualified names `N1.N2.A` and `N1.N2.B`. Because the two declarations contribute to the same declaration space, it would have been an error if each contained a declaration of a member with the same name.

9.3 Extern Aliases

An *extern-alias-directive* introduces an identifier that serves as an alias for a namespace. The specification of the aliased namespace is external to the source code of the program and also applies to nested namespaces of the aliased namespace.

extern-alias-directives:
    extern-alias-directive
    extern-alias-directives *extern-alias-directive*

extern-alias-directive:
    `extern alias` *identifier* `;`

The scope of an *extern-alias-directive* extends over the *using-directives*, *global-attributes*, and *namespace-member-declarations* of its immediately containing compilation unit or namespace body.

Within a compilation unit or namespace body that contains an *extern-alias-directive*, the identifier introduced by the *extern-alias-directive* can be used to reference the aliased namespace. It is a compile-time error for the *identifier* to be the word `global`.

An *extern-alias-directive* makes an alias available within a particular compilation unit or namespace body, but it does not contribute any new members to the underlying declaration space. In other words, an *extern-alias-directive* is not transitive, but rather affects only the compilation unit or namespace body in which it occurs.

www.it-ebooks.info

The following program declares and uses two extern aliases, `X` and `Y`, each of which represents the root of a distinct namespace hierarchy:

```
extern alias X;
extern alias Y;

class Test
{
    X::N.A a;
    X::N.B b1;
    Y::N.B b2;
    Y::N.C c;
}
```

The program declares the existence of the extern aliases `X` and `Y`, but the actual definitions of the aliases are external to the program. The identically named `N.B` classes can now be referenced as `X.N.B` and `Y.N.B`, or, using the namespace alias qualifier, `X::N.B` and `Y::N.B`. An error occurs if a program declares an extern alias for which no external definition is provided.

9.4 Using Directives

Using directives facilitate the use of namespaces and types defined in other namespaces. Using directives impact the name resolution process of *namespace-or-type-name*s (§3.8) and *simple-name*s (§7.6.2), but unlike declarations, using directives do not contribute new members to the underlying declaration spaces of the compilation units or namespaces within which they are used.

using-directives:
    using-directive
    using-directives using-directive

using-directive:
    using-alias-directive
    using-namespace-directive

A *using-alias-directive* (§9.4.1) introduces an alias for a namespace or type.

A *using-namespace-directive* (§9.4.2) imports the type members of a namespace.

The scope of a *using-directive* extends over the *namespace-member-declarations* of its immediately containing compilation unit or namespace body. The scope of a *using-directive* specifically does not include its peer *using-directive*s. Thus peer *using-directive*s do not affect one another, and the order in which they are written is insignificant.

www.it-ebooks.info

9.4.1 Using Alias Directives

A *using-alias-directive* introduces an identifier that serves as an alias for a namespace or type within the immediately enclosing compilation unit or namespace body.

using-alias-directive:
    `using` *identifier* `=` *namespace-or-type-name* `;`

Within member declarations in a compilation unit or namespace body that contains a *using-alias-directive*, the identifier introduced by the *using-alias-directive* can be used to reference the given namespace or type. For example:

```
namespace N1.N2
{
    class A { }
}

namespace N3
{
    using A = N1.N2.A;

    class B : A { }
}
```

In this example, within the member declarations in the `N3` namespace, `A` is an alias for `N1.N2.A`, and thus class `N3.B` derives from class `N1.N2.A`. The same effect can be obtained by creating an alias `R` for `N1.N2` and then referencing `R.A`:

```
namespace N3
{
    using R = N1.N2;

    class B : R.A { }
}
```

The *identifier* of a *using-alias-directive* must be unique within the declaration space of the compilation unit or namespace that immediately contains the *using-alias-directive*. For example:

```
namespace N3
{
    class A { }
}

namespace N3
{
    using A = N1.N2.A;        // Error: A already exists
}
```

In this example, `N3` already contains a member `A`, so it is a compile-time error for a *using-alias-directive* to use that identifier. Likewise, it is a compile-time error for two or more

www.it-ebooks.info

*using-alias-directive*s in the same compilation unit or namespace body to declare aliases by the same name.

A *using-alias-directive* makes an alias available within a particular compilation unit or namespace body, but it does not contribute any new members to the underlying declaration space. In other words, a *using-alias-directive* is not transitive, but rather affects only the compilation unit or namespace body in which it occurs. In the example

```
namespace N3
{
    using R = N1.N2;
}

namespace N3
{
    class B : R.A { }              // Error: R unknown
}
```

the scope of the *using-alias-directive* that introduces `R` extends only to member declarations in the namespace body in which it is contained, so `R` is unknown in the second namespace declaration. However, placing the *using-alias-directive* in the containing compilation unit causes the alias to become available within both namespace declarations:

```
using R = N1.N2;

namespace N3
{
    class B: R.A {}
}

namespace N3
{
    class C: R.A {}
}
```

Just like regular members, names introduced by *using-alias-directive*s are hidden by similarly named members in nested scopes. In the example

```
using R = N1.N2;

namespace N3
{
    class R {}

    class B: R.A {}                // Error: R has no member A
}
```

the reference to `R.A` in the declaration of `B` causes a compile-time error because `R` refers to `N3.R`, not `N1.N2`.

The order in which *using-alias-directive*s are written has no significance, and resolution of the *namespace-or-type-name* referenced by a *using-alias-directive* is not affected by the

www.it-ebooks.info

using-alias-directive itself or by other *using-directive*s in the immediately containing compilation unit or namespace body. In other words, the *namespace-or-type-name* of a *using-alias-directive* is resolved as if the immediately containing compilation unit or namespace body had no *using-directive*s. A *using-alias-directive* may, however, be affected by *extern-alias-directives* in the immediately containing compilation unit or namespace body. In the example

```
namespace N1.N2 { }

namespace N3
{
    extern alias E;

    using R1 = E.N;           // Okay

    using R2 = N1;            // Okay

    using R3 = N1.N2;         // Okay

    using R4 = R2.N2;         // Error: R2 unknown
}
```

the last *using-alias-directive* results in a compile-time error because it is not affected by the first *using-alias-directive*. The first *using-alias-directive* does not result in an error since the scope of the extern alias `E` includes the *using-alias-directive*.

A *using-alias-directive* can create an alias for any namespace or type, including the namespace within which it appears and any namespace or type nested within that namespace.

Accessing a namespace or type through an alias yields exactly the same result as accessing that namespace or type through its declared name. For example, given

```
namespace N1.N2
{
    class A { }
}

namespace N3
{
    using R1 = N1;
    using R2 = N1.N2;

    class B
    {
        N1.N2.A a;            // Refers to N1.N2.A
        R1.N2.A b;            // Refers to N1.N2.A
        R2.A c;               // Refers to N1.N2.A
    }
}
```

the names `N1.N2.A`, `R1.N2.A`, and `R2.A` are equivalent and all refer to the class whose fully qualified name is `N1.N2.A`.

www.it-ebooks.info

Using aliases can name a closed constructed type, but cannot name an unbound generic type declaration without supplying type arguments. For example:

```
namespace N1
{
    class A<T>
    {
        class B {}
    }
}

namespace N2
{
    using W = N1.A;          // Error: cannot name unbound generic type

    using X = N1.A.B;        // Error, cannot name unbound generic type

    using Y = N1.A<int>;     // Okay: can name closed constructed type

    using Z<T> = N1.A<T>;    // Error: using alias cannot have type parameters
}
```

9.4.2 Using Namespace Directives

A *using-namespace-directive* imports the types contained in a namespace into the immediately enclosing compilation unit or namespace body, enabling the identifier of each type to be used without qualification.

using-namespace-directive:
    `using` *namespace-name* `;`

Within member declarations in a compilation unit or namespace body that contains a *using-namespace-directive*, the types contained in the given namespace can be referenced directly. For example:

```
namespace N1.N2
{
    class A { }
}

namespace N3
{
    using N1.N2;

    class B : A { }
}
```

In this example, within the member declarations in the `N3` namespace, the type members of `N1.N2` are directly available, and thus class `N3.B` derives from class `N1.N2.A`.

www.it-ebooks.info

A *using-namespace-directive* imports the types contained in the given namespace, but specifically does not import nested namespaces. In the example

```
namespace N1.N2
{
    class A { }
}

namespace N3
{
    using N1;

    class B : N2.A { }        // Error: N2 unknown
}
```

the *using-namespace-directive* imports the types contained in `N1`, but not the namespaces nested in `N1`. Thus the reference to `N2.A` in the declaration of `B` results in a compile-time error because no members named `N2` are in scope.

Unlike a *using-alias-directive*, a *using-namespace-directive* may import types whose identifiers are already defined within the enclosing compilation unit or namespace body. In effect, names imported by a *using-namespace-directive* are hidden by similarly named members in the enclosing compilation unit or namespace body. For example:

```
namespace N1.N2
{
    class A { }

    class B { }
}

namespace N3
{
    using N1.N2;

    class A { }
}
```

Here, within the member declarations in the `N3` namespace, `A` refers to `N3.A` rather than `N1.N2.A`.

When more than one namespace imported by *using-namespace-directive*s in the same compilation unit or namespace body contain types by the same name, references to that name are considered ambiguous. In the example

```
namespace N1
{
    class A { }
}

namespace N2
{
    class A { }
}
```


www.it-ebooks.info

```
namespace N3
{
    using N1;

    using N2;

    class B : A { }           // Error: A is ambiguous
}
```

both `N1` and `N2` contain a member `A`, and because `N3` imports both, referencing `A` in `N3` is a compile-time error. In this situation, the conflict can be resolved either through qualification of references to `A` or by introducing a *using-alias-directive* that picks a particular `A`. For example:

```
namespace N3
{
    using N1;

    using N2;

    using A = N1.A;

    class B : A { }            // A means N1.A
}
```

Like a *using-alias-directive*, a *using-namespace-directive* does not contribute any new members to the underlying declaration space of the compilation unit or namespace, but rather affects only the compilation unit or namespace body in which it appears.

The *namespace-name* referenced by a *using-namespace-directive* is resolved in the same way as the *namespace-or-type-name* referenced by a *using-alias-directive*. Thus *using-namespace-directive*s in the same compilation unit or namespace body do not affect one another and can be written in any order.

9.5 Namespace Members

A *namespace-member-declaration* is either a *namespace-declaration* (§9.2) or a *type-declaration* (§9.6).

namespace-member-declarations:
    namespace-member-declaration
    namespace-member-declarations namespace-member-declaration

namespace-member-declaration:
    namespace-declaration
    type-declaration

www.it-ebooks.info

A compilation unit or a namespace body can contain *namespace-member-declarations*, and such declarations contribute new members to the underlying declaration space of the containing compilation unit or namespace body.

9.6 Type Declarations

A *type-declaration* is a *class-declaration* (§10.1), a *struct-declaration* (§11.1), an *interface-declaration* (§13.1), an *enum-declaration* (§14.1), or a *delegate-declaration* (§15.1).

type-declaration:
class-declaration
struct-declaration
interface-declaration
enum-declaration
delegate-declaration

A *type-declaration* can occur as a top-level declaration in a compilation unit or as a member declaration within a namespace, class, or struct.

When a type declaration for a type `T` occurs as a top-level declaration in a compilation unit, the fully qualified name of the newly declared type is simply `T`. When a type declaration for a type `T` occurs within a namespace, class, or struct, the fully qualified name of the newly declared type is `N.T`, where `N` is the fully qualified name of the containing namespace, class, or struct.

A type declared within a class or struct is called a nested type (§10.3.8).

The permitted access modifiers and the default access for a type declaration depend on the context in which the declaration takes place (§3.5.1):

- Types declared in compilation units or namespaces can have `public` or `internal` access. The default is `internal` access.
- Types declared in classes can have `public`, `protected internal`, `protected`, `internal`, or `private` access. The default is `private` access.
- Types declared in structs can have `public`, `internal`, or `private` access. The default is `private` access.

9.7 Namespace Alias Qualifiers

The ***namespace alias qualifier*** `::` makes it possible to guarantee that type name lookups are unaffected by the introduction of new types and members. The namespace alias

www.it-ebooks.info

qualifier always appears between two identifiers, referred to as the left-hand and right-hand identifiers. Unlike the regular `.` qualifier, the left-hand identifier of the `::` qualifier is looked up only as an extern or using alias.

A *qualified-alias-member* is defined as follows:

qualified-alias-member:
identifier `::` *identifier* *type-argument-list*$_{opt}$

A *qualified-alias-member* can be used as a *namespace-or-type-name* (§3.8) or as the left operand in a *member-access* (§7.6.4).

A *qualified-alias-member* has one of two forms:

- `N::I<`A_1`, ..., `A_K`>`, where `N` and `I` represent identifiers, and `<`A_1`, ..., `A_K`>` is a type argument list. (`K` is always at least one.)
- `N::I`, where `N` and `I` represent identifiers. (In this case, `K` is considered to be zero.)

Using this notation, the meaning of a *qualified-alias-member* is determined as follows:

- If `N` is the identifier `global`, then the global namespace is searched for `I`:
  - If the global namespace contains a namespace named `I` and `K` is zero, then the *qualified-alias-member* refers to that namespace.
  - Otherwise, if the global namespace contains a nongeneric type named `I` and `K` is zero, then the *qualified-alias-member* refers to that type.
  - Otherwise, if the global namespace contains a type named `I` that has `K` type parameters, then the *qualified-alias-member* refers to that type constructed with the given type arguments.
  - Otherwise, the *qualified-alias-member* is undefined and a compile-time error occurs.
- Otherwise, starting with the namespace declaration (§9.2) immediately containing the *qualified-alias-member* (if any), continuing with each enclosing namespace declaration (if any), and ending with the compilation unit containing the *qualified-alias-member*, the following steps are evaluated until an entity is located:
  - If the namespace declaration or compilation unit contains a *using-alias-directive* that associates `N` with a type, then the *qualified-alias-member* is undefined and a compile-time error occurs.
  - Otherwise, if the namespace declaration or compilation unit contains an *extern-alias-directive* or *using-alias-directive* that associates `N` with a namespace, then:
    - If the namespace associated with `N` contains a namespace named `I` and `K` is zero, then the *qualified-alias-member* refers to that namespace.

www.it-ebooks.info

  - Otherwise, if the namespace associated with `N` contains a nongeneric type named `I` and `K` is zero, then the *qualified-alias-member* refers to that type.
  - Otherwise, if the namespace associated with `N` contains a type named `I` that has `K` type parameters, then the *qualified-alias-member* refers to that type constructed with the given type arguments.
  - Otherwise, the *qualified-alias-member* is undefined and a compile-time error occurs.
- Otherwise, the *qualified-alias-member* is undefined and a compile-time error occurs.

Note that using the namespace alias qualifier with an alias that references a type causes a compile-time error. Also note that if the identifier `N` is `global`, then lookup is performed in the global namespace, even if there is a using alias associating `global` with a type or namespace.

9.7.1 Uniqueness of Aliases

Each compilation unit and namespace body has a separate declaration space for extern aliases and using aliases. Thus, while the name of an extern alias or using alias must be unique within the set of extern aliases and using aliases declared in the immediately containing compilation unit or namespace body, an alias is permitted to have the same name as a type or namespace as long as it is used only with the `::` qualifier.

In the example

```
namespace N
{
    public class A { }

    public class B { }
}

namespace N
{
    using A = System.IO;

    class X
    {
        A.Stream s1;              // Error: A is ambiguous

        A::Stream s2;             // Okay
    }
}
```

the name `A` has two possible meanings in the second namespace body because both the class `A` and the using alias `A` are in scope. For this reason, use of `A` in the qualified name `A.Stream` is ambiguous and causes a compile-time error to occur. However, use of `A` with the `::` qualifier is not an error because `A` is looked up only as a namespace alias.

www.it-ebooks.info

10. Classes

A class is a data structure that may contain data members (constants and fields), function members (methods, properties, events, indexers, operators, instance constructors, destructors, and static constructors), and nested types. Class types support inheritance, a mechanism whereby a derived class can extend and specialize a base class.

10.1 Class Declarations

A *class-declaration* is a *type-declaration* (§9.6) that declares a new class.

class-declaration:
attributes$_{opt}$ *class-modifiers*$_{opt}$ `partial`$_{opt}$ `class` *identifier* *type-parameter-list*$_{opt}$
class-base$_{opt}$ *type-parameter-constraints-clauses*$_{opt}$ *class-body* `;`$_{opt}$

A *class-declaration* consists of an optional set of *attributes* (§17), followed by an optional set of *class-modifiers* (§10.1.1), followed by an optional `partial` modifier, followed by the keyword `class` and an *identifier* that names the class, followed by an optional *type-parameter-list* (§10.1.3), followed by an optional *class-base* specification (§10.1.4), followed by an optional set of *type-parameter-constraints-clauses* (§10.1.5), followed by a *class-body* (§10.1.6), optionally followed by a semicolon.

A class declaration cannot supply *type-parameter-constraints-clauses* unless it also supplies a *type-parameter-list*.

A class declaration that supplies a *type-parameter-list* is a ***generic class declaration***. Additionally, any class nested inside a generic class declaration or a generic struct declaration is itself a generic class declaration, since type parameters for the containing type must be supplied to create a constructed type.

10.1.1 Class Modifiers

A *class-declaration* may optionally include a sequence of class modifiers:

class-modifiers:
class-modifier
class-modifiers *class-modifier*

www.it-ebooks.info

class-modifier:
    `new`
    `public`
    `protected`
    `internal`
    `private`
    `abstract`
    `sealed`
    `static`

It is a compile-time error for the same modifier to appear multiple times in a class declaration.

The `new` modifier is permitted on nested classes. It specifies that the class hides an inherited member by the same name, as described in §10.3.4. It is a compile-time error for the `new` modifier to appear on a class declaration that is not a nested class declaration.

The `public`, `protected`, `internal`, and `private` modifiers control the accessibility of the class. Depending on the context in which the class declaration occurs, some of these modifiers may not be permitted (§3.5.1).

The `abstract`, `sealed`, and `static` modifiers are discussed in the following sections.

10.1.1.1 *Abstract Classes*

The `abstract` modifier is used to indicate that a class is incomplete and that it is intended to be used only as a base class. An abstract class differs from a nonabstract class in the following ways:

- An abstract class cannot be instantiated directly, and it is a compile-time error to use the `new` operator on an abstract class. While it is possible to have variables and values whose compile-time types are abstract, such variables and values will necessarily either be `null` or contain references to instances of nonabstract classes derived from the abstract types.
- An abstract class is permitted (but not required) to contain abstract members.
- An abstract class cannot be sealed.

When a nonabstract class is derived from an abstract class, the nonabstract class must include actual implementations of all inherited abstract members, thereby overriding those abstract members. In the example

```
abstract class A
{
    public abstract void F();
}

abstract class B: A
```


www.it-ebooks.info

```
{
    public void G() {}
}

class C: B
{
    public override void F() {
        // Actual implementation of F
    }
}
```

the abstract class `A` introduces an abstract method `F`. Class `B` introduces an additional method `G`, but since it doesn't provide an implementation of `F`, `B` must also be declared abstract. Class `C` overrides `F` and provides an actual implementation. Since there are no abstract members in `C`, `C` is permitted (but not required) to be nonabstract.

10.1.1.2 *Sealed Classes*

The `sealed` modifier is used to prevent derivation from a class. A compile-time error occurs if a sealed class is specified as the base class of another class.

A sealed class cannot also be an abstract class.

The `sealed` modifier is primarily used to prevent unintended derivation, but it also enables certain runtime optimizations. In particular, because a sealed class is known to never have any derived classes, it is possible to transform virtual function member invocations on sealed class instances into nonvirtual invocations.

JON SKEET The choice to make classes unsealed (but methods nonvirtual) by default has always been a hotly disputed one. I agree with the maxim "Design for inheritance or prohibit it," but there are arguments both ways. The odd point is how powerful a default is: Even though I usually *choose* to seal classes if I think about it, it's all too easy to ignore the choice entirely. It obviously doesn't affect what can be expressed, but I'm certain it affects the code that is actually produced.

JESSE LIBERTY I will take the risk of disagreeing with Jon.

It is demonstrably false that you can accurately anticipate what will be needed even months in advance. Thus good programming practice would dictate letting your design emerge, building little or nothing before it is explicitly needed, and avoiding closing off avenues you believe you'll never need.

Sealed is a minefield laid down to state, "You'll never need to go here." I don't think you can know that or should presume to try.

www.it-ebooks.info

10.1.1.3 *Static Classes*

The `static` modifier is used to mark the class being declared as a ***static class***. A static class cannot be instantiated, cannot be used as a type, and can contain only static members. Only a static class can contain declarations of extension methods (§10.6.9).

A static class declaration is subject to the following restrictions:

- A static class may not include a `sealed` or `abstract` modifier. Note, however, that since a static class cannot be instantiated or derived from, it behaves as if it was both sealed and abstract.
- A static class may not include a *class-base* specification (§10.1.4) and cannot explicitly specify a base class or a list of implemented interfaces. A static class implicitly inherits from type `object`.
- A static class can only contain static members (§10.3.7). Note that constants and nested types are classified as static members.
- A static class cannot have members with `protected` or `protected internal` declared accessibility.

It is a compile-time error to violate any of these restrictions.

A static class has no instance constructors. It is not possible to declare an instance constructor in a static class, and no default instance constructor (§10.11.4) is provided for a static class.

The members of a static class are not automatically static, and the member declarations must explicitly include a `static` modifier (except for constants and nested types). When a class is nested within a static outer class, the nested class is not a static class unless it explicitly includes a `static` modifier.

> **MAREK SAFAR** A sealed class with a private constructor had to be used to simulate static classes in C# 1.0.That is no longer needed, as static classes offer a more elegant way to express this intention, plus the benefit of many compiler checks for operations that are not allowed in a static context.

10.1.1.3.1 Referencing Static Class Types

A *namespace-or-type-name* (§3.8) is permitted to reference a static class if

- The *namespace-or-type-name* is the `T` in a *namespace-or-type-name* of the form `T.I`, or
- The *namespace-or-type-name* is the `T` in a *typeof-expression* (§7.5.11) of the form `typeof(T)`.

www.it-ebooks.info

A *primary-expression* (§7.5) is permitted to reference a static class if

- The *primary-expression* is the `E` in a *member-access* (§7.5.4) of the form `E.I`.

In any other context, it is a compile-time error to reference a static class. For example, it is an error for a static class to be used as a base class, a constituent type (§10.3.8) of a member, a generic type argument, or a type parameter constraint. Likewise, a static class cannot be used in an array type, a pointer type, a `new` expression, a cast expression, an `is` expression, an `as` expression, a `sizeof` expression, or a default value expression.

10.1.2 partial Modifier

The `partial` modifier is used to indicate that this *class-declaration* is a partial type declaration. Multiple partial type declarations with the same name within an enclosing namespace or type declaration combine to form one type declaration, following the rules specified in §10.2.

Having the declaration of a class distributed over separate segments of program text can be useful if these segments are produced or maintained in different contexts. For instance, one part of a class declaration may be machine generated, whereas the other is authored manually. Textual separation of the two prevents updates by one from conflicting with updates by the other.

> **CHRIS SELLS** I love partial classes' ability to split that part that's machine generated from the part that's human generated. Unfortunately, you can abuse partial classes by splitting a class across more than two files. As a reader of code, I find this practice extremely difficult to follow, and I strongly discourage it.

10.1.3 Type Parameters

A type parameter is a simple identifier that denotes a placeholder for a type argument supplied to create a constructed type. A type parameter is a formal placeholder for a type that will be supplied later. By contrast, a type argument (§4.4.1) is the actual type that is substituted for the type parameter when a constructed type is created.

type-parameter-list:
    `<` *type-parameters* `>`

type-parameters:
    attributes$_{opt}$ *type-parameter*
    type-parameters `,` *attributes*$_{opt}$ *type-parameter*

type-parameter:
    identifier

www.it-ebooks.info

Each type parameter in a class declaration defines a name in the declaration space (§3.3) of that class. Thus it cannot have the same name as another type parameter or a member declared in that class. A type parameter also cannot have the same name as the type itself.

10.1.4 Class Base Specification

A class declaration may include a *class-base* specification, which defines the direct base class of the class and the interfaces (§13) directly implemented by the class.

class-base:
    `:` *class-type*
    `:` *interface-type-list*
    `:` *class-type* `,` *interface-type-list*

interface-type-list:
    interface-type
    interface-type-list `,` *interface-type*

The base class specified in a class declaration can be a constructed class type (§4.4). A base class cannot be a type parameter on its own, although it can involve the type parameters that are in scope.

```
class Extend<V>: V {}       // Error: type parameter used as base class
```

> **BILL WAGNER** I wish this restriction could be removed. It would be a great way to create mixins. I realize it's a very difficult problem because `V` may contain any arbitrary methods and properties. Depending on any concrete type used for `V` in a closed generic type, `Extend<V>` may not compile.

10.1.4.1 *Base Classes*

When a *class-type* is included in the *class-base*, it specifies the direct base class of the class being declared. If a class declaration has no *class-base*, or if the *class-base* lists only interface types, the direct base class is assumed to be `object`. A class inherits members from its direct base class, as described in §10.3.3.

In the example

```
class A {}
class B: A {}
```

class `A` is said to be the direct base class of `B`, and `B` is said to be derived from `A`. Since `A` does not explicitly specify a direct base class, its direct base class is implicitly `object`.

For a constructed class type, if a base class is specified in the generic class declaration, the base class of the constructed type is obtained by substituting, for each *type-parameter* in

www.it-ebooks.info

the base class declaration, the corresponding *type-argument* of the constructed type. Given the generic class declarations

```
class B<U,V> {...}

class G<T>: B<string,T[]> {...}
```

the base class of the constructed type `G<int>` would be `B<string,int[]>`.

The direct base class of a class type must be at least as accessible as the class type itself (§3.5.2). For example, it is a compile-time error for a `public` class to derive from a `private` or `internal` class.

The direct base class of a class type must not be any of the following types: `System.Array`, `System.Delegate`, `System.MulticastDelegate`, `System.Enum`, or `System.ValueType`. Furthermore, a generic class declaration cannot use `System.Attribute` as a direct or indirect base class.

While determining the meaning of the direct base class specification `A` of a class `B`, the direct base class of `B` is temporarily assumed to be `object`. Intuitively, this ensures that the meaning of a base class specification cannot recursively depend on itself. The example

```
class A<T> {
   public class B{}
}

class C : A<C.B> {}
```

is in error since in the base class specification `A<C.B>`, the direct base class of `C` is considered to be `object`; hence (by the rules of §3.8), `C` is not considered to have a member `B`.

The base classes of a class type are the direct base class and its base classes. In other words, the set of base classes is the transitive closure of the direct base class relationship. Referring to the example above, the base classes of `B` are `A` and `object`. In the example

```
class A {...}

class B<T>: A {...}

class C<T>: B<IComparable<T>> {...}

class D<T>: C<T[]> {...}
```

the base classes of `D<int>` are `C<int[]>`, `B<IComparable<int[]>>`, `A`, and `object`.

Except for class `object`, every class type has exactly one direct base class. The `object` class has no direct base class and is the ultimate base class of all other classes.

When a class `B` derives from a class `A`, it is a compile-time error for `A` to depend on `B`. A class ***directly depends on*** its direct base class (if any) and ***directly depends on*** the class within which it is immediately nested (if any). Given this definition, the complete set of classes

www.it-ebooks.info

upon which a class depends is the reflexive and transitive closure of the ***directly depends on*** relationship.

> **VLADIMIR RESHETNIKOV** For the purposes of this rule, type arguments, if any, are ignored. For instance, although `A<T>` and `A<A<T>>` are different types, the following declaration is still invalid:
>
> ```
> class A<T> : A<A<T>> { }
> ```
>
> Conversely, it is perfectly valid for a class to appear within a type argument for a constructed type specified as its base class:
>
> ```
> class A<T> { }
> class B : A<B[]> { } // Okay
> ```


The example

```
class A: A {}
```

is erroneous because the class depends on itself. Likewise, the example

```
class A: B {}

class B: C {}

class C: A {}
```

is in error because the classes circularly depend on themselves. Finally, the example

```
class A: B.C {}

class B: A
{
    public class C {}
}
```

results in a compile-time error because `A` depends on `B.C` (its direct base class), which depends on `B` (its immediately enclosing class), which circularly depends on `A`.

Note that a class does not depend on the classes that are nested within it. In the example

```
class A
{
    class B: A {}
}
```

`B` depends on `A` (because `A` is both its direct base class and its immediately enclosing class), but `A` does not depend on `B` (because `B` is neither a base class nor an enclosing class of `A`). Thus the example is valid.

It is not possible to derive from a `sealed` class. In the example

www.it-ebooks.info

```
sealed class A {}
class B: A {}         // Error: cannot derive from a sealed class
```

class `B` is in error because it attempts to derive from the `sealed` class `A`.

10.1.4.2 *Interface Implementations*

A *class-base* specification may include a list of interface types, in which case the class is said to directly implement the given interface types. Interface implementations are discussed further in §13.4.

10.1.5 Type Parameter Constraints

Generic type and method declarations can optionally specify type parameter constraints by including *type-parameter-constraints-clause*s.

type-parameter-constraints-clauses:
    type-parameter-constraints-clause
    type-parameter-constraints-clauses *type-parameter-constraints-clause*

type-parameter-constraints-clause:
    `where` *type-parameter* `:` *type-parameter-constraints*

type-parameter-constraints:
    primary-constraint
    secondary-constraints
    constructor-constraint
    primary-constraint `,` *secondary-constraints*
    primary-constraint `,` *constructor-constraint*
    secondary-constraints `,` *constructor-constraint*
    primary-constraint `,` *secondary-constraints* `,` *constructor-constraint*

primary-constraint:
    class-type
    `class`
    `struct`

secondary-constraints:
    interface-type
    type-parameter
    secondary-constraints `,` *interface-type*
    secondary-constraints `,` *type-parameter*

constructor-constraint:
    `new` `(` `)`

www.it-ebooks.info

Each *type-parameter-constraints-clause* consists of the token `where`, followed by the name of a type parameter, followed by a colon and the list of constraints for that type parameter. There can be at most one `where` clause for each type parameter, and the `where` clauses can be listed in any order. Like the `get` and `set` tokens in a property accessor, the `where` token is not a keyword.

The list of constraints given in a `where` clause can include any of the following components, in this order: a single primary constraint, one or more secondary constraints, and the constructor constraint, `new()`.

A primary constraint can be a class type or the ***reference type constraint*** `class` or the ***value type constraint*** `struct`. A secondary constraint can be a *type-parameter* or *interface-type*.

The reference type constraint specifies that a type argument used for the type parameter must be a reference type. All class types, interface types, delegate types, array types, and type parameters known to be a reference type (as defined below) satisfy this constraint.

The value type constraint specifies that a type argument used for the type parameter must be a non-nullable value type. All non-nullable struct types, enum types, and type parameters having the value type constraint satisfy this constraint. Although it is classified as a value type, a nullable type (§4.1.10) does not satisfy the value type constraint. A type parameter having the value type constraint cannot also have the *constructor-constraint*.

> **BILL WAGNER** At one point, I found this set of constraints very limiting. I wanted to do meta-programming with generics, being able to specify any arbitrary set of members as constraints—things like other constructor signatures, or operators. Now that C# 3.0 has lambdas and more natural support for the use of methods as parameters, that feeling is gone. By specifying delegate signatures as type parameters, or as parameters to methods, C# programmers can achieve almost anything.

Pointer types are never allowed to be type arguments and are not considered to satisfy either the reference type or value type constraints.

If a constraint is a class type, an interface type, or a type parameter, that type specifies a minimal "base type" that every type argument used for that type parameter must support. Whenever a constructed type or generic method is used, the type argument is checked against the constraints on the type parameter at compile time. The type argument supplied must satisfy the conditions described in §4.4.4.

A *class-type* constraint must satisfy the following rules:

- The type must be a class type.
- The type must not be `sealed`.

www.it-ebooks.info

- The type must not be one of the following types: `System.Array`, `System.Delegate`, `System.Enum`, or `System.ValueType`.

> **JON SKEET** I know of no reason to prohibit `System.Enum` or `System.Delegate` as the type here—although it would certainly be more friendly to be able to write "`where T : enum`" than "`where T : struct, Enum`", which would be the most common use. Although C# is specified separately from the CLI, you might suspect that this is a CLI limitation—but it's not. Indeed, ECMA-335 explicitly lists this and other constraints that are prohibited in C#. Perhaps this restriction will be removed from a future version of the language; there are plenty of situations where it would be useful.

- The type must not be `object`. Because all types derive from `object`, such a constraint would have no effect if it were permitted.
- At most one constraint for a given type parameter can be a class type.

A type specified as an *interface-type* constraint must satisfy the following rules:

- The type must be an interface type.
- A type must not be specified more than once in a given `where` clause.

In either case, the constraint can involve any of the type parameters of the associated type or method declaration as part of a constructed type, and can involve the type being declared.

Any class or interface type specified as a type parameter constraint must be at least as accessible (§3.5.4) as the generic type or method being declared.

A type specified as a *type-parameter* constraint must satisfy the following rules:

- The type must be a type parameter.
- A type must not be specified more than once in a given `where` clause.

In addition, there must be no cycles in the dependency graph of type parameters, where dependency is a transitive relation defined as follows:

- If a type parameter `T` is used as a constraint for type parameter `S`, then `S` ***depends on*** `T`.
- If a type parameter `S` depends on a type parameter `T` and `T` depends on a type parameter `U`, then `S` ***depends on*** `U`.

Given this relation, it is a compile-time error for a type parameter to depend on itself (directly or indirectly).

www.it-ebooks.info

Any constraints must be consistent among dependent type parameters. If type parameter `S` depends on type parameter `T`, then

- `T` must not have the value type constraint. Otherwise, `T` is effectively sealed so `S` would be forced to be the same type as `T`, eliminating the need for two type parameters.
- If `S` has the value type constraint, then `T` must not have a *class-type* constraint.
- If `S` has a *class-type* constraint `A` and `T` has a *class-type* constraint `B`, then there must be an identity conversion or implicit reference conversion from `A` to `B` or an implicit reference conversion from `B` to `A`.
- If `S` also depends on type parameter `U` and `U` has a *class-type* constraint `A` and `T` has a *class-type* constraint `B`, then there must be an identity conversion or implicit reference conversion from `A` to `B` or an implicit reference conversion from `B` to `A`.

It is valid for `S` to have the value type constraint and `T` to have the reference type constraint. Effectively, this limits `T` to the types `System.Object`, `System.ValueType`, `System.Enum`, and any interface type.

If the `where` clause for a type parameter includes a constructor constraint (which has the form `new()`), it is possible to use the `new` operator to create instances of the type (§7.6.10.1). Any type argument used for a type parameter with a constructor constraint must have a public parameterless constructor (this constructor implicitly exists for any value type) or be a type parameter having the value type constraint or constructor constraint (see §10.1.5 for details).

The following are examples of constraints:

```
interface IPrintable
{
    void Print();
}

interface IComparable<T>
{
    int CompareTo(T value);
}

interface IKeyProvider<T>
{
    T GetKey();
}

class Printer<T> where T: IPrintable {...}

class SortedList<T> where T: IComparable<T> {...}
```


www.it-ebooks.info

```
class Dictionary<K,V>
    where K: IComparable<K>
    where V: IPrintable, IKeyProvider<K>, new()
{
    ...
}
```

The following example is in error because it causes a circularity in the dependency graph of the type parameters:

```
class Circular<S,T>
    where S: T
    where T: S      // Error: circularity in dependency graph
{
    ...
}
```

The following examples illustrate additional invalid situations:

```
class Sealed<S,T>
    where S: T
    where T: struct   // Error: T is sealed
{
    ...
}

class A {...}

class B {...}

class Incompat<S,T>
    where S: A, T
    where T: B      // Error: incompatible class-type constraints
{
    ...
}

class StructWithClass<S,T,U>
    where S: struct, T
    where T: U
    where U: A      // Error: A incompatible with struct
{
    ...
}
```

The ***effective base class*** of a type parameter `T` is defined as follows:

- If `T` has no primary constraints or type parameter constraints, its effective base class is `object`.
- If `T` has the value type constraint, its effective base class is `System.ValueType`.
- If `T` has a *class-type* constraint `C` but no *type-parameter* constraints, its effective base class is `C`.

www.it-ebooks.info

- If `T` has no *class-type* constraint but has one or more *type-parameter* constraints, its effective base class is the most encompassed type (§6.4.2) in the set of effective base classes of its *type-parameter* constraints. The consistency rules ensure that such a most encompassed type exists.
- If `T` has both a *class-type* constraint and one or more *type-parameter* constraints, its effective base class is the most encompassed type (§6.4.2) in the set consisting of the *class-type* constraint of `T` and the effective base classes of its *type-parameter* constraints. The consistency rules ensure that such a most encompassed type exists.
- If `T` has the reference type constraint but no *class-type* constraints, its effective base class is `object`.

For the purpose of these rules, if T has a constraint `V` that is a *value-type*, use instead the most specific base type of `V` that is a *class-type*. This can never happen in an explicitly given constraint, but may occur when the constraints of a generic method are implicitly inherited by an overriding method declaration or an explicit implementation of an interface method.

These rules ensure that the effective base class is always a *class-type*.

ERIC LIPPERT For example, suppose you have

```
class B<T> { public virtual void M<U>() where U : T {} }
class D : B<DateTime> { public override void M<V>() }
```

Then the effective base class of `V` is the class type `System.ValueType`, not the struct type `DateTime`. Similarly, if instead of `DateTime`, we had `DateTime[]`, then the effective base class would be the class type `System.Array`, not the array type `DateTime[]`.

The ***effective interface set*** of a type parameter `T` is defined as follows:

- If `T` has no *secondary-constraints*, its effective interface set is empty.
- If `T` has *interface-type* constraints but no *type-parameter* constraints, its effective interface set is its set of *interface-type* constraints.
- If `T` has no *interface-type* constraints but has *type-parameter* constraints, its effective interface set is the union of the effective interface sets of its *type-parameter* constraints.
- If `T` has both *interface-type* constraints and *type-parameter* constraints, its effective interface set is the union of its set of *interface-type* constraints and the effective interface sets of its *type-parameter* constraints.

www.it-ebooks.info

A type parameter is ***known to be a reference type*** if it has the reference type constraint or its effective base class is not `object` or `System.ValueType`.

Values of a constrained type parameter type can be used to access the instance members implied by the constraints. In the example

```
interface IPrintable
{
    void Print();
}

class Printer<T> where T: IPrintable
{
    void PrintOne(T x) {
        x.Print();
    }
}
```

the methods of `IPrintable` can be invoked directly on `x` because `T` is constrained to always implement `IPrintable`.

10.1.6 Class Body

The *class-body* of a class defines the members of that class.

class-body:
`{` *class-member-declarations*$_{opt}$ `}`

10.2 Partial Types

A type declaration can be split across multiple ***partial type declarations***. The type declaration is constructed from its parts by following the rules in this section, whereupon it is treated as a single declaration during the remainder of the compile-time and runtime processing of the program.

A *class-declaration*, *struct-declaration*, or *interface-declaration* represents a partial type declaration if it includes a `partial` modifier. Note that `partial` is not a keyword, and acts as a modifier only if it appears immediately before one of the keywords `class`, `struct`, or `interface` in a type declaration, or before the type `void` in a method declaration. In other contexts, it can be used as a normal identifier.

Each part of a partial type declaration must include a `partial` modifier. It must have the same name and be declared in the same namespace or type declaration as the other parts. The `partial` modifier indicates that additional parts of the type declaration may exist elsewhere, but the existence of such additional parts is not a requirement; it is valid for a type with a single declaration to include the `partial` modifier.

www.it-ebooks.info

All parts of a partial type must be compiled together such that the parts can be merged at compile time into a single type declaration. Partial types specifically do not allow already compiled types to be extended.

Nested types may be declared in multiple parts by using the `partial` modifier. Typically, the containing type is declared using `partial` as well, and each part of the nested type is declared in a different part of the containing type.

The `partial` modifier is not permitted on delegate or enum declarations.

BILL WAGNER This feature was clearly added to support code generators, but it has many other uses. I've put nested classes in separate compilation units, and broken apart classes on other logical boundaries. However, in the general case, splitting classes simply to let multiple developers work on the same class is not advised.

BRAD ABRAMS The traditional model employed by visual software design tools is to provide some user interface that developers use to express their intent; the tool then generates source code based on that intent. This approach is a time-tested and widely applicable model. The data design time, ASP.NET design time, and WinForms design time, for example, all use this basic model. However, this model does present some challenges—namely, developers often need to tweak, modify, or extend the code generated by the tool. Editing the generated code directly is a popular solution, but it has the major disadvantage of making the visual design tool unusable. Subclassing from the generated code is another approach, but it is often complicated by type issues. Partial types and methods allow for a class to be partially generated by a design tool and partially customized to better suit a given scenario.

10.2.1 Attributes

The attributes of a partial type are determined by combining, in an unspecified order, the attributes of each of the parts. If an attribute is placed on multiple parts, it is equivalent to specifying the attribute multiple times on the type. For example, the two parts

```
[Attr1, Attr2("hello")]
partial class A {}

[Attr3, Attr2("goodbye")]
partial class A {}
```


www.it-ebooks.info

are equivalent to the following declaration:

```
[Attr1, Attr2("hello"), Attr3, Attr2("goodbye")]
class A {}
```

Attributes on type parameters may be combined in a similar fashion.

10.2.2 Modifiers

When a partial type declaration includes an accessibility specification (the `public`, `protected`, `internal`, and `private` modifiers), it must agree with all other parts that include an accessibility specification. If no part of a partial type includes an accessibility specification, the type is given the appropriate default accessibility (§3.5.1).

If one or more partial declarations of a nested type include a `new` modifier, no warning is reported if the nested type hides an inherited member (§3.7.1.2).

If one or more partial declarations of a class include an `abstract` modifier, the class is considered abstract (§10.1.1.1). Otherwise, the class is considered nonabstract.

If one or more partial declarations of a class include a `sealed` modifier, the class is considered sealed (§10.1.1.2). Otherwise, the class is considered unsealed.

Note that a class cannot be both abstract and sealed.

When the `unsafe` modifier is used on a partial type declaration, only that particular part is considered an unsafe context (§18.1).

10.2.3 Type Parameters and Constraints

If a generic type is declared in multiple parts, each part must state the type parameters. Each part must have the same number of type parameters, and the same name for each type parameter, in order.

When a partial generic type declaration includes constraints (`where` clauses), the constraints must agree with all other parts that include constraints. Specifically, each part that includes constraints must have constraints for the same set of type parameters, and for each type parameter the sets of primary, secondary, and constructor constraints must be equivalent. Two sets of constraints are equivalent if they contain the same members. If no part of a partial generic type specifies type parameter constraints, the type parameters are considered unconstrained.

www.it-ebooks.info

The example

```
partial class Dictionary<K,V>
    where K: IComparable<K>
    where V: IKeyProvider<K>, IPersistable
{
    ...
}

partial class Dictionary<K,V>
    where V: IPersistable, IKeyProvider<K>
    where K: IComparable<K>
{
    ...
}

partial class Dictionary<K,V>
{
    ...
}
```

is correct because those parts that include constraints (the first two) effectively specify the same set of primary, secondary, and constructor constraints for the same set of type parameters, respectively.

BILL WAGNER When possible, I prefer specifying the type parameters and constraints on all copies. This technique improves readability.

10.2.4 Base Class

When a partial class declaration includes a base class specification, it must agree with all other parts that include a base class specification. If no part of a partial class includes a base class specification, the base class becomes `System.Object` (§10.1.4.1).

10.2.5 Base Interfaces

The set of base interfaces for a type declared in multiple parts is the union of the base interfaces specified on each part. A particular base interface may be named only once on each part, but it is permitted for multiple parts to name the same base interface(s). There must be only one implementation of the members of any given base interface.

In the example

```
partial class C: IA, IB {...}

partial class C: IC {...}

partial class C: IA, IB {...}
```

the set of base interfaces for class `C` is `IA`, `IB`, and `IC`.

www.it-ebooks.info

Typically, each part provides an implementation of the interface(s) declared on that part; however, this is not a requirement. A part may provide the implementation for an interface declared on a different part:

```
partial class X
{
    int IComparable.CompareTo(object o) {...}
}

partial class X: IComparable
{
    ...
}
```

10.2.6 Members

With the exception of partial methods (§10.2.7), the set of members of a type declared in multiple parts is simply the union of the set of members declared in each part. The bodies of all parts of the type declaration share the same declaration space (§3.3), and the scope of each member (§3.7) extends to the bodies of all the parts. The accessibility domain of any member always includes all the parts of the enclosing type; a `private` member declared in one part is freely accessible from another part. It is a compile-time error to declare the same member in more than one part of the type, unless that member is a type with the `partial` modifier.

> **BILL WAGNER** Logically, you can think of the following example as one long source file, except you don't know the order in which the class contents are combined.

```
partial class A
{
    int x;                  // Error: cannot declare x more than once

    partial class Inner     // Okay: Inner is a partial type
    {
        int y;
    }
}

partial class A
{
    int x;                  // Error: cannot declare x more than once

    partial class Inner     // Okay: Inner is a partial type
    {
        int z;
    }
}
```

The ordering of members within a type is rarely significant to C# code, but may be significant when interfacing with other languages and environments. In these cases, the ordering of members within a type declared in multiple parts is undefined.

www.it-ebooks.info

JON SKEET One area where ordering is important in C# is static and instance variable initializers: They are guaranteed to be executed in the "textual order" (§10.5.5) in which they appear in the class. This is no great loss, as relying on such ordering is usually a bad idea: A class that breaks when you just reorder declarations is too brittle to start with.

It wouldn't be unreasonable to expect that members declared *within the same part* are handled in the obvious order, although the specification doesn't explicitly guarantee it.

10.2.7 Partial Methods

Partial methods can be defined in one part of a type declaration and implemented in another. The implementation is optional; if no part implements the partial method, the partial method declaration and all calls to it are removed from the type declaration resulting from the combination of the parts.

VLADIMIR RESHETNIKOV A partial method can be declared only in a partial class or partial struct. It cannot be declared in a nonpartial type or in an interface.

Partial methods cannot define access modifiers, but are implicitly `private`. Their return type must be `void`, and their parameters cannot have the `out` modifier. The identifier `partial` is recognized as a special keyword in a method declaration only if it appears right before the `void` type; otherwise, it can be used as a normal identifier. A partial method cannot explicitly implement interface methods.

There are two kinds of partial method declarations: If the body of the method declaration is a semicolon, the declaration is said to be a ***defining partial method declaration***. If the body is given as a *block*, the declaration is said to be an ***implementing partial method declaration***. Across the parts of a type declaration, there can be only one defining partial method declaration with a given signature, and there can be only one implementing partial method declaration with a given signature. If an implementing partial method declaration is given, a corresponding defining partial method declaration must exist, and the declarations must match as specified in the following:

- The declarations must have the same modifiers (although not necessarily in the same order), method name, number of type parameters, and number of parameters.
- Corresponding parameters in the declarations must have the same modifiers (although not necessarily in the same order) and the same types (modulo differences in type parameter names).

www.it-ebooks.info

- Corresponding type parameters in the declarations must have the same constraints (modulo differences in type parameter names).

An implementing partial method declaration can appear in the same part as the corresponding defining partial method declaration.

Only a defining partial method participates in overload resolution. Thus, whether or not an implementing declaration is given, invocation expressions may resolve to invocations of the partial method. Because a partial method always returns `void`, such invocation expressions will always be expression statements. Furthermore, because a partial method is implicitly `private`, such statements will always occur within one of the parts of the type declaration within which the partial method is declared.

If no part of a partial type declaration contains an implementing declaration for a given partial method, any expression statement invoking it is simply removed from the combined type declaration. Thus the invocation expression, including any constituent expressions, has no effect at runtime. The partial method itself is also removed and will not be a member of the combined type declaration.

If an implementing declaration exists for a given partial method, the invocations of the partial methods are retained. The partial method gives rise to a method declaration similar to the implementing partial method declaration except for the following:

- The `partial` modifier is not included.
- The attributes in the resulting method declaration are the combined attributes of the defining and the implementing partial method declaration in unspecified order. Duplicates are not removed.
- The attributes on the parameters of the resulting method declaration are the combined attributes of the corresponding parameters of the defining and the implementing partial method declaration in unspecified order. Duplicates are not removed.

If a defining declaration but not an implementing declaration is given for a partial method M, the following restrictions apply:

- It is a compile-time error to create a delegate to method (§7.6.10.5).
- It is a compile-time error to refer to `M` inside an anonymous function that is converted to an expression tree type (§6.5.2).
- Expressions occurring as part of an invocation of `M` do not affect the definite assignment state (§5.3), which can potentially lead to compile-time errors.
- `M` cannot be the entry point for an application (§3.1).

www.it-ebooks.info

Partial methods are useful for allowing one part of a type declaration to customize the behavior of another part—for example, one that is generated by a tool. Consider the following partial class declaration:

```
partial class Customer
{
    string name;

    public string Name {

        get { return name; }

        set {
            OnNameChanging(value);
            name = value;
            OnNameChanged();
        }

    }

    partial void OnNameChanging(string newName);

    partial void OnNameChanged();
}
```

If this class is compiled without any other parts, the defining partial method declarations and their invocations will be removed, and the resulting combined class declaration will be equivalent to the following:

```
class Customer
{
    string name;

    public string Name {

        get { return name; }

        set { name = value; }
    }
}
```

Assume that another part is given, however, which provides implementing declarations of the partial methods:

```
partial class Customer
{
    partial void OnNameChanging(string newName)
    {
        Console.WriteLine("Changing " + name + " to " + newName);
    }

    partial void OnNameChanged()
    {
        Console.WriteLine("Changed to " + name);
    }
}
```

Then the resulting combined class declaration will be equivalent to the following:

www.it-ebooks.info

```
class Customer
{
    string name;

    public string Name {

        get { return name; }

        set {
            OnNameChanging(value);
            name = value;
            OnNameChanged();
        }

    }

    void OnNameChanging(string newName)
    {
        Console.WriteLine("Changing " + name + " to " + newName);
    }

    void OnNameChanged()
    {
        Console.WriteLine("Changed to " + name);
    }
}
```

CHRIS SELLS Before the introduction of partial methods, the pattern was to use virtual methods like so:

```
class Base {
    public void Foo() { HelpWithFoo(); ... }
        // Dynamically bound
    protected virtual void HelpWithFoo() {}
        // Do nothing in the base
}
class Derived : Base {
    protected override void HelpWithFoo() { ... }
}
```

This approach required machine-generated code to use less efficient dynamically bound virtual methods to "call over" to the other part of the partial class, even though the implementation was available at compile time. With partial methods, the pattern becomes the more efficient:

```
partial class MyClass {             // Machine generated
    public void Foo() { HelpWithFoo(); ... }
        // Statically bound
    partial void HelpWithFoo();
        // Declare the partial method
}
partial class MyClass {             // Human generated
    partial void HelpWithFoo() { ... }
        // Implement the partial method
}
```


www.it-ebooks.info

> **ERIC LIPPERT** Chris's point is well taken. I would add that there are more costs to consider than the extra couple of nanoseconds required to perform the virtual call. Suppose you have a machine-generated class with potentially hundreds of points of extensibility, where the machine-generated code wants to call a helper method defined in the user-generated side. If the author of the user-generated side wants to implement only one of those hundreds, all the code for the call sites and all the metadata for the methods are still generated. Partial methods are truly "pay for play": You take on the additional code size only for the extensibility points you actually use.

10.2.8 Name Binding

Although each part of an extensible type must be declared within the same namespace, the parts are typically written within different namespace declarations. Thus different `using` directives (§9.4) may be present for each part. When interpreting simple names (§7.5.2) within one part, only the `using` directives of the namespace declaration(s) enclosing that part are considered. This may result in the same identifier having different meanings in different parts:

```
namespace N
{
    using List = System.Collections.ArrayList;

    partial class A
    {
        List x;     // x has type System.Collections.ArrayList
    }
}

namespace N
{
    using List = Widgets.LinkedList;

    partial class A
    {
        List y;     // y has type Widgets.LinkedList
    }
}
```

10.3 Class Members

The members of a class consist of the members introduced by its *class-member-declaration*s and the members inherited from the direct base class.

class-member-declarations:
    class-member-declaration
    class-member-declarations class-member-declaration

www.it-ebooks.info

class-member-declaration:
constant-declaration
field-declaration
method-declaration
property-declaration
event-declaration
indexer-declaration
operator-declaration
constructor-declaration
destructor-declaration
static-constructor-declaration
type-declaration

The members of a class type are divided into the following categories:

- Constants, which represent constant values associated with the class (§10.4).
- Fields, which are the variables of the class (§10.5).
- Methods, which implement the computations and actions that can be performed by the class (§10.6).
- Properties, which define named characteristics and the actions associated with reading and writing those characteristics (§10.7).
- Events, which define notifications that can be generated by the class (§10.8).
- Indexers, which permit instances of the class to be indexed in the same way (syntactically) as arrays (§10.9).
- Operators, which define the expression operators that can be applied to instances of the class (§10.10).
- Instance constructors, which implement the actions required to initialize instances of the class (§10.11).
- Destructors, which implement the actions to be performed before instances of the class are permanently discarded (§10.13).
- Static constructors, which implement the actions required to initialize the class itself (§10.12).
- Types, which represent the types that are local to the class (§10.3.8).

Members that can contain executable code are collectively known as the *function members* of the class type. The function members of a class type are the methods, properties, events, indexers, operators, instance constructors, destructors, and static constructors of that class type.

www.it-ebooks.info

A *class-declaration* creates a new declaration space (§3.3), and the *class-member-declarations* immediately contained by the *class-declaration* introduce new members into this declaration space. The following rules apply to *class-member-declaration*s:

- Instance constructors, destructors, and static constructors must have the same name as the immediately enclosing class. All other members must have names that differ from the name of the immediately enclosing class.
- The names of constants, fields, properties, events, or types must differ from the names of all other members declared in the same class.
- The name of a method must differ from the names of all other nonmethods declared in the same class. In addition, the signature (§3.6) of a method must differ from the signatures of all other methods declared in the same class, and two methods declared in the same class may not have signatures that differ solely by `ref` and `out`.
- The signature of an instance constructor must differ from the signatures of all other instance constructors declared in the same class, and two constructors declared in the same class may not have signatures that differ solely by `ref` and `out`.
- The signature of an indexer must differ from the signatures of all other indexers declared in the same class.
- The signature of an operator must differ from the signatures of all other operators declared in the same class.

The inherited members of a class type (§10.3.3) are not part of the declaration space of a class. Thus a derived class is allowed to declare a member with the same name or signature as an inherited member (which, in effect, hides the inherited member).

10.3.1 The Instance Type

Each class declaration has an associated bound type (§4.4.3), known as the ***instance type***. For a generic class declaration, the instance type is formed by creating a constructed type (§4.4) from the type declaration, with each of the supplied type arguments being the corresponding type parameter. Since the instance type uses the type parameters, it can be used only where the type parameters are in scope—that is, inside the class declaration. The instance type is the type of `this` for code written inside the class declaration. For nongeneric classes, the instance type is simply the declared class. The following shows several class declarations along with their instance types:

```
class A<T>                  // Instance type: A<T>
{
    class B {}              // Instance type: A<T>.B

    class C<U> {}           // Instance type: A<T>.C<U>
}

class D {}                  // Instance type: D
```


www.it-ebooks.info

10.3.2 Members of Constructed Types

The non-inherited members of a constructed type are obtained by substituting, for each *type-parameter* in the member declaration, the corresponding *type-argument* of the constructed type. The substitution process is based on the semantic meaning of type declarations, and is not simply textual substitution.

For example, given the generic class declaration

```
class Gen<T,U>
{
    public T[,] a;

    public void G(int i, T t, Gen<U,T> gt) {...}

    public U Prop { get {...} set {...} }

    public int H(double d) {...}
}
```

the constructed type `Gen<int[],IComparable<string>>` has the following members:

```
public int[,][] a;

public void G(int i, int[] t, Gen<IComparable<string>,int[]> gt) {...}

public IComparable<string> Prop { get {...} set {...} }

public int H(double d) {...}
```

The type of the member `a` in the generic class declaration `Gen` is "two-dimensional array of `T`," so the type of the member `a` in the constructed type above is "two-dimensional array of one-dimensional array of `int`," or `int[,][]`.

Within instance function members, the type of `this` is the instance type (§10.3.1) of the containing declaration.

JON SKEET This treatment of generics via substitution leads to obvious tricky situations. What would the following code do?

```
public class Puzzle<T> {
  public void Method(int i) {}
  public void Method(T t) {}
}
...
new Puzzle<int>().Method(10);
```

The "substitution" results in two methods with exactly the same signature—so which one is called by the `Method(10)` expression? I readily admit that I wouldn't know the answer without looking it up or trying it. I strongly suggest that such ambiguity should be avoided wherever possible. In my experience, it's most likely to occur with a pair of indexers that can look an item up either by position (an `int`) or key (a type parameter).

www.it-ebooks.info

ERIC LIPPERT In an early design of generics in C# 2.0, the language designers considered making it illegal to even *declare* a class that could *possibly* be constructed so as to produce a signature ambiguity like the one Jon describes. Unfortunately, that restriction then makes illegal certain patterns that seem obviously desirable:

```
class C<T> { public C(T t) { ... } public C(Stream serializedState) { ... } }
```

Should it be illegal to declare `C<T>` just because someone might someday make a `C<Stream>`? This seems like overkill.

Even so, it is an extremely poor programming practice to create types that can have ambiguous signatures under construction and then actually make such constructions. Doing so can, in some contrived cases, expose implementation-defined behavior in the CLR.

All members of a generic class can use type parameters from any enclosing class, either directly or as part of a constructed type. When a particular closed constructed type (§4.4.2) is used at runtime, each use of a type parameter is replaced with the actual type argument supplied to the constructed type. For example:

```
class C<V>
{
    public V f1;
    public C<V> f2 = null;

    public C(V x) {
        this.f1 = x;
        this.f2 = this;
    }
}

class Application
{
    static void Main() {
        C<int> x1 = new C<int>(1);
        Console.WriteLine(x1.f1);           // Prints 1

        C<double> x2 = new C<double>(3.1415);
        Console.WriteLine(x2.f1);           // Prints 3.1415
    }
}
```

10.3.3 Inheritance

A class ***inherits*** the members of its direct base class type. Inheritance means that a class implicitly contains all members of its direct base class type, except for the instance constructors, destructors, and static constructors of the base class. Some important aspects of inheritance are profiled here:

www.it-ebooks.info

- Inheritance is transitive. If `C` is derived from `B`, and `B` is derived from `A`, then `C` inherits the members declared in `B` as well as the members declared in `A`.
- A derived class *extends* its direct base class. A derived class can add new members to those it inherits, but it cannot remove the definition of an inherited member.
- Instance constructors, destructors, and static constructors are not inherited, but all other members are, regardless of their declared accessibility (§3.5). However, depending on their declared accessibility, inherited members might not be accessible in a derived class.

JON SKEET Occasionally, developers ask why instance constructors aren't inherited. My view is that only the particular class knows which information it needs to create a valid instance. For example, if instance constructors were inherited, all classes would have to have a parameterless constructor (because `object` does). What would it mean to create a new `FileStream` without specifying a filename or handle?

- A derived class can ***hide*** (§3.7.1.2) inherited members by declaring new members with the same name or signature. Note, however, that hiding an inherited member does not remove that member—it merely makes that member inaccessible directly through the derived class.
- An instance of a class contains a set of all instance fields declared in the class and its base classes, and an implicit conversion (§6.1.6) exists from a derived class type to any of its base class types. Thus a reference to an instance of some derived class can be treated as a reference to an instance of any of its base classes.
- A class can declare virtual methods, properties, and indexers, and derived classes can override the implementation of these function members. This enables classes to exhibit polymorphic behavior wherein the actions performed by a function member invocation varies depending on the runtime type of the instance through which that function member is invoked.

The inherited members of a constructed class type are the members of the immediate base class type (§10.1.4.1), which is found by substituting the type arguments of the constructed type for each occurrence of the corresponding type parameters in the *base-class-specification*. These members, in turn, are transformed by substituting, for each *type-parameter* in the member declaration, the corresponding *type-argument* of the *base-class-specification*.

```
class B<U>
{
    public U F(long index) {...}
}

class D<T>: B<T[]>
{
    public T G(string s) {...}
}
```


www.it-ebooks.info

In the above example, the constructed type `D<int>` has a non-inherited member `public int G(string s)` obtained by substituting the type argument `int` for the type parameter `T`. `D<int>` also has an inherited member from the class declaration `B`. This inherited member is determined by first determining the base class type `B<int[]>` of `D<int>` by substituting `int` for `T` in the base class specification `B<T[]>`. Then, as a type argument to `B`, `int[]` is substituted for `U` in `public U F(long index)`, yielding the inherited member `public int[] F(long index)`.

10.3.4 The new Modifier

A *class-member-declaration* is permitted to declare a member with the same name or signature as an inherited member. When this occurs, the derived class member is said to ***hide*** the base class member. Hiding an inherited member is not considered an error, but it does cause the compiler to issue a warning. To suppress the warning, the declaration of the derived class member can include a `new` modifier to indicate that the derived member is intended to hide the base member. This topic is discussed further in §3.7.1.2.

If a `new` modifier is included in a declaration that doesn't hide an inherited member, a warning to that effect is issued. This warning is suppressed by removing the `new` modifier.

10.3.5 Access Modifiers

A *class-member-declaration* can have any one of the five possible kinds of declared accessibility (§3.5.1): `public`, `protected internal`, `protected`, `internal`, or `private`. Except for the `protected internal` combination, it is a compile-time error to specify more than one access modifier. When a *class-member-declaration* does not include any access modifiers, `private` is assumed.

10.3.6 Constituent Types

Types that are used in the declaration of a member are called the constituent types of that member. Possible constituent types are the type of a constant, field, property, event, or indexer; the return type of a method or operator; and the parameter types of a method, indexer, operator, or instance constructor. The constituent types of a member must be at least as accessible as that member itself (§3.5.4).

10.3.7 Static and Instance Members

Members of a class are either ***static members*** or ***instance members***. Generally speaking, it is useful to think of static members as belonging to class types and instance members as belonging to objects (instances of class types).

www.it-ebooks.info

When a field, method, property, event, operator, or constructor declaration includes a `static` modifier, it declares a static member. In addition, a constant or type declaration implicitly declares a static member. Static members have the following characteristics:

- When a static member `M` is referenced in a *member-access* (§7.6.4) of the form `E.M`, `E` must denote a type containing `M`. It is a compile-time error for `E` to denote an instance.
- A static field identifies exactly one storage location to be shared by all instances of a given closed class type. No matter how many instances of a given closed class type are created, there is only ever one copy of a static field.
- A static function member (method, property, event, operator, or constructor) does not operate on a specific instance, and it is a compile-time error to refer to `this` in such a function member.

When a field, method, property, event, indexer, constructor, or destructor declaration does not include a `static` modifier, it declares an instance member. (An instance member is sometimes called a nonstatic member.) Instance members have the following characteristics:

- When an instance member `M` is referenced in a *member-access* (§7.6.4) of the form `E.M`, `E` must denote an instance of a type containing `M`. It is a binding-time error for `E` to denote a type.
- Every instance of a class contains a separate set of all instance fields of the class.
- An instance function member (method, property, indexer, instance constructor, or destructor) operates on a given instance of the class, and this instance can be accessed as `this` (§7.6.7).

The following example illustrates the rules for accessing static and instance members:

```
class Test
{
    int x;
    static int y;

    void F() {
        x = 1;                  // Okay: same as this.x = 1
        y = 1;                  // Okay: same as Test.y = 1
    }

    static void G() {
        x = 1;                  // Error: cannot access this.x
        y = 1;                  // Okay: same as Test.y = 1
    }
```


www.it-ebooks.info

```
        static void Main() {
            Test t = new Test();
            t.x = 1;                // Okay
            t.y = 1;                // Error: cannot access static
                                    // member through instance
            Test.x = 1;             // Error: cannot access instance member
                                    // through type
            Test.y = 1;             // Okay
        }
    }
```

The `F` method shows that in an instance function member, a *simple-name* (§7.6.2) can be used to access both instance members and static members. The `G` method shows that in a static function member, it is a compile-time error to access an instance member through a *simple-name*. The `Main` method shows that in a *member-access* (§7.6.4), instance members must be accessed through instances, and static members must be accessed through types.

10.3.8 Nested Types

A type declared within a class or struct declaration is called a ***nested type***. A type that is declared within a compilation unit or namespace is called a ***non-nested type***.

> **JON SKEET** I've heard the phrase "top-level type" used more often than "non-nested type"—and I've no doubt spread this nonstandard terminology myself. Considering that the vast majority of types are non-nested, it feels like the descriptive term should be something positive, rather than just an absence of nesting. That's my excuse, anyway, and others are welcome to borrow it.

In the example

```
using System;

class A
{
    class B
    {
        static void F() {
            Console.WriteLine("A.B.F");
        }
    }
}
```

class `B` is a nested type because it is declared within class `A`, and class `A` is a non-nested type because it is declared within a compilation unit.

www.it-ebooks.info

BRAD ABRAMS I am not a big fan of publicly accessible nested types in a reusable library. Discoverability for nested types isn't great. Even more importantly, we already have a group mechanism—namespaces—that should be used to indicate related types.

10.3.8.1 *Fully Qualified Names*

The fully qualified name (§3.8.1) for a nested type is `S.N`, where `S` is the fully qualified name of the type in which type `N` is declared.

10.3.8.2 *Declared Accessibility*

Non-nested types can have `public` or `internal` declared accessibility and have `internal` declared accessibility by default. Nested types can have these forms of declared accessibility as well, plus one or more additional forms of declared accessibility, depending on whether the containing type is a class or struct:

- A nested type that is declared in a class can have any of five forms of declared accessibility (`public`, `protected internal`, `protected`, `internal`, or `private`) and, like other class members, defaults to `private` declared accessibility.
- A nested type that is declared in a struct can have any of three forms of declared accessibility (`public`, `internal`, or `private`) and, like other struct members, defaults to `private` declared accessibility.

The example

```
public class List
{
    // Private data structure
    private class Node
    {
        public object Data;
        public Node Next;

        public Node(object data, Node next) {
            this.Data = data;
            this.Next = next;
        }
    }

    private Node first = null;
    private Node last = null;

    // Public interface
    public void AddToFront(object o) {...}
```


www.it-ebooks.info

```
    public void AddToBack(object o) {...}

    public object RemoveFromFront() {...}

    public object RemoveFromBack() {...}

    public int Count { get {...} }
}
```

declares a private nested class `Node`.

10.3.8.3 *Hiding*

A nested type may hide (§3.7.1) a base member. The `new` modifier is permitted on nested type declarations so that hiding can be expressed explicitly. The example

```
class Base
{
    public static void M() {
        Console.WriteLine("Base.M");
    }
}

class Derived: Base
{
    new public class M
    {
        public static void F() {
            Console.WriteLine("Derived.M.F");
        }
    }
}

class Test
{
    static void Main() {
        Derived.M.F();
    }
}
```

shows a nested class `M` that hides the method `M` defined in `Base`.

10.3.8.4 *this Access*

A nested type and its containing type do not have a special relationship with regard to *this-access* (§7.6.7). Specifically, `this` within a nested type cannot be used to refer to instance members of the containing type. In cases where a nested type needs access to the instance members of its containing type, access can be provided by providing the

www.it-ebooks.info

`this` for the instance of the containing type as a constructor argument for the nested type. The following example

```
using System;

class C
{
    int i = 123;

    public void F() {
        Nested n = new Nested(this);
        n.G();
    }

    public class Nested
    {
        C this_c;

        public Nested(C c) {
            this_c = c;
        }

        public void G() {
            Console.WriteLine(this_c.i);
        }
    }
}

class Test
{
    static void Main() {
        C c = new C();
        c.F();
    }
}
```

shows this technique. An instance of `C` creates an instance of `Nested` and passes its own `this` to `Nested`'s constructor to provide subsequent access to `C`'s instance members.

JON SKEET This section of the specification may seem strange at first sight: Why bother to specify what you *can't* do? Why would anyone expect to be able to get at an instance of the containing type from an instance of a nested type? Well, in Java, that's exactly how inner classes do work. To achieve C#-like nested type behavior, you need to declare the nested type as `static`—which is nothing like a `static` class in C#!

www.it-ebooks.info

10.3.8.5 ***Access to Private and Protected Members of the Containing Type***

A nested type has access to all of the members that are accessible to its containing type, including members of the containing type that have `private` and `protected` declared accessibility. The example

```
using System;

class C
{
    private static void F() {
        Console.WriteLine("C.F");
    }

    public class Nested
    {
        public static void G() {
            F();
        }
    }
}

class Test
{
    static void Main() {
        C.Nested.G();
    }
}
```

shows a class `C` that contains a nested class `Nested`. Within `Nested`, the method `G` calls the `static` method `F` defined in `C`, and `F` has `private` declared accessibility.

> **JON SKEET** Again, this behavior is different from that in Java, where an outer type has access to the `private` members of a nested type, but not vice versa. I find the C# approach to be more sensible. It also allows for some nice patterns, such as creating an abstract class with a `private` constructor: All the derived classes have to be nested within the abstract class. Oddly enough, I find the main use case for this behavior is emulating Java enums.

A nested type also may access `protected` members defined in a base type of its containing type. In the example

```
using System;

class Base
{
    protected void F() {
        Console.WriteLine("Base.F");
    }
}
```


www.it-ebooks.info

```
class Derived: Base
{
    public class Nested
    {
        public void G() {
            Derived d = new Derived();
            d.F();            // Okay
        }
    }
}

class Test
{
    static void Main() {
        Derived.Nested n = new Derived.Nested();
        n.G();
    }
}
```

the nested class `Derived.Nested` accesses the `protected` method `F` defined in `Derived`'s base class, `Base`, by calling through an instance of `Derived`.

10.3.8.6 *Nested Types in Generic Classes*

A generic class declaration can contain nested type declarations. The type parameters of the enclosing class can be used within the nested types. A nested type declaration can contain additional type parameters that apply only to the nested type.

Every type declaration contained within a generic class declaration is implicitly a generic type declaration. When writing a reference to a type nested within a generic type, the containing constructed type, including its type arguments, must be named. However, from within the outer class, the nested type can be used without qualification; the instance type of the outer class can be implicitly used when constructing the nested type. The following example shows three different—and correct—ways to refer to a constructed type created from `Inner`; the first two are equivalent:

```
class Outer<T>
{
    class Inner<U>
    {
        public static void F(T t, U u) {...}
    }

    static void F(T t) {
        Outer<T>.Inner<string>.F(t, "abc");    // These two statements
        Inner<string>.F(t, "abc");             // have the same effect

        Outer<int>.Inner<string>.F(3, "abc");  // This type is different

        Outer.Inner<string>.F(t, "abc");       // Error: Outer needs type arg
    }
}
```


www.it-ebooks.info

Although it is bad programming style, a type parameter in a nested type can hide a member or type parameter declared in the outer type:

```
class Outer<T>
{
    class Inner<T>                          // Valid: hides Outer's T
    {
        public T t;                         // Refers to Inner's T
    }
}
```

10.3.9 Reserved Member Names

To facilitate the underlying C# runtime implementation, for each source member declaration that is a property, event, or indexer, the implementation must reserve two method signatures based on the kind of the member declaration, its name, and its type. It is a compile-time error for a program to declare a member whose signature matches one of these reserved signatures, even if the underlying runtime implementation does not make use of these reservations.

The reserved names do not introduce declarations; thus they do not participate in member lookup. However, a declaration's associated reserved method signatures do participate in inheritance (§10.3.3) and can be hidden with the `new` modifier (§10.3.4).

The reservation of these names serves three purposes:

- It allows the underlying implementation to use an ordinary identifier as a method name for `get` or `set` access to the C# language feature.
- It allows other languages to interoperate use of an ordinary identifier as a method name for `get` or `set` access to the C# language feature.
- It helps ensure that the source accepted by one conforming compiler is accepted by another, by making the specifics of reserved member names consistent across all C# implementations.

The declaration of a destructor (§10.13) also causes a signature to be reserved (§10.3.9.4).

10.3.9.1 *Member Names Reserved for Properties*

For a property `P` (§10.7) of type `T`, the following signatures are reserved:

```
T get_P();
void set_P(T value);
```

Both signatures are reserved, even if the property is read-only or write-only.

www.it-ebooks.info

> **BILL WAGNER** Even though you could write methods that are named `get_<something>` and `set_<something>`, if `<something>` is not a property name, this technique is not recommended. Avoiding it will minimize the need to update the code in the future.

In the example

```
using System;

class A
{
    public int P {
        get { return 123; }
    }
}

class B: A
{
    new public int get_P() {
        return 456;
    }

    new public void set_P(int value) {
    }
}

class Test
{
    static void Main() {
        B b = new B();
        A a = b;
        Console.WriteLine(a.P);
        Console.WriteLine(b.P);
        Console.WriteLine(b.get_P());
    }
}
```

a class `A` defines a read-only property `P`, thus reserving signatures for `get_P` and `set_P` methods. The class `B` derives from `A` and hides both of these reserved signatures. The example produces the following output:

```
123
123
456
```


www.it-ebooks.info

10.3.9.2 *Member Names Reserved for Events*

For an event `E` (§10.8) of delegate type `T`, the following signatures are reserved:

```
void add_E(T handler);
void remove_E(T handler);
```

10.3.9.3 *Member Names Reserved for Indexers*

For an indexer (§10.9) of type `T` with parameter-list `L`, the following signatures are reserved:

```
T get_Item(L);
void set_Item(L, T value);
```

Both signatures are reserved, even if the indexer is read-only or write-only. Furthermore, the member name `Item` is reserved.

10.3.9.4 *Member Names Reserved for Destructors*

For a class containing a destructor (§10.13), the following signature is reserved:

```
void Finalize();
```


10.4 Constants

A ***constant*** is a class member that represents a constant value—a value that can be computed at compile time. A *constant-declaration* introduces one or more constants of a given type.

constant-declaration:
attributes$_{opt}$ *constant-modifiers*$_{opt}$ `const` *type* *constant-declarators* `;`

constant-modifiers:
constant-modifier
constant-modifiers *constant-modifier*

constant-modifier:
`new`
`public`
`protected`
`internal`
`private`

constant-declarators:
constant-declarator
constant-declarators `,` *constant-declarator*

www.it-ebooks.info

constant-declarator:
    identifier `=` *constant-expression*

A *constant-declaration* may include a set of *attributes* (§17), a `new` modifier (§10.3.4), and a valid combination of the four access modifiers (§10.3.5). The attributes and modifiers apply to all of the members declared by the *constant-declaration*. Even though constants are considered static members, a *constant-declaration* neither requires nor allows a `static` modifier. It is an error for the same modifier to appear multiple times in a constant declaration.

The *type* of a *constant-declaration* specifies the type of the members introduced by the declaration. The type is followed by a list of *constant-declarator*s, each of which introduces a new member. A *constant-declarator* consists of an *identifier* that names the member, followed by an "`=`" token, followed by a *constant-expression* (§7.19) that gives the value of the member.

The *type* specified in a constant declaration must be either `sbyte`, `byte`, `short`, `ushort`, `int`, `uint`, `long`, `ulong`, `char`, `float`, `double`, `decimal`, `bool`, `string`, an *enum-type*, or a *reference-type*. Each *constant-expression* must yield a value of the target type or of a type that can be converted to the target type by an implicit conversion (§6.1).

JON SKEET The inclusion of `decimal` here is interesting when targeting the Common Language Infrastructure. All of the other types have literal representations within the CLI, whereas `decimal` doesn't. The Microsoft C# compiler achieves the appropriate behavior by creating a static read-only field declaration, albeit decorated with `DecimalConstantAttribute`. A similar attribute exists for `DateTime`, but C# doesn't allow `DateTime` constants.

The *type* of a constant must be at least as accessible as the constant itself (§3.5.4).

The value of a constant is obtained in an expression using a *simple-name* (§7.6.2) or a *member-access* (§7.6.4).

A constant can itself participate in a *constant-expression*. Thus a constant may be used in any construct that requires a *constant-expression*. Examples of such constructs include `case` labels, `goto case` statements, `enum` member declarations, attributes, and other constant declarations.

As described in §7.19, a *constant-expression* is an expression that can be fully evaluated at compile time. Since the only way to create a non-null value of a *reference-type* other than `string` is to apply the `new` operator, and since the `new` operator is not permitted in a *constant-expression*, the only possible value for constants of *reference-types* other than `string` is `null`.

www.it-ebooks.info

When a symbolic name for a constant value is desired, but when the type of that value is not permitted in a constant declaration, or when the value cannot be computed at compile time by a *constant-expression*, a `readonly` field (§10.5.2) may be used instead.

A constant declaration that declares multiple constants is equivalent to multiple declarations of single constants with the same attributes, modifiers, and type. For example,

```
class A
{
    public const double X = 1.0, Y = 2.0, Z = 3.0;
}
```

is equivalent to

```
class A
{
    public const double X = 1.0;
    public const double Y = 2.0;
    public const double Z = 3.0;
}
```


CHRIS SELLS From a readability point of view, I find multiple declarations on the same line—whether constants or nonconstants, static or instance, fields or local variables—to be difficult to decipher. I prefer to place only a single declaration on each line, ideally keeping the locality of reference in mind.

Constants are permitted to depend on other constants within the same program as long as the dependencies are not of a circular nature. The compiler automatically arranges to evaluate the constant declarations in the appropriate order. In the example

```
class A
{
    public const int X = B.Z + 1;
    public const int Y = 10;
}

class B
{
    public const int Z = A.Y + 1;
}
```

the compiler first evaluates `A.Y`, then evaluates `B.Z`, and finally evaluates `A.X`, producing the values `10`, `11`, and `12`, in that order. Constant declarations may depend on constants from other programs, but such dependencies are only possible in one direction. Referring to the example above, if `A` and `B` were declared in separate programs, it would be possible for `A.X` to depend on `B.Z`, but `B.Z` could then not simultaneously depend on `A.Y`.

www.it-ebooks.info

10.5 Fields

A ***field*** is a member that represents a variable associated with an object or class. A *field-declaration* introduces one or more fields of a given type.

field-declaration:
    attributes$_{opt}$ *field-modifiers*$_{opt}$ *type* *variable-declarators* `;`

field-modifiers:
    field-modifier
    field-modifiers *field-modifier*

field-modifier:
    `new`
    `public`
    `protected`
    `internal`
    `private`
    `static`
    `readonly`
    `volatile`

variable-declarators:
    variable-declarator
    variable-declarators `,` *variable-declarator*

variable-declarator:
    identifier
    identifier `=` *variable-initializer*

variable-initializer:
    expression
    array-initializer

A *field-declaration* may include a set of *attributes* (§17), a `new` modifier (§10.3.4), a valid combination of the four access modifiers (§10.3.5), and a `static` modifier (§10.5.1). In addition, a *field-declaration* may include a `readonly` modifier (§10.5.2) or a `volatile` modifier (§10.5.3), but not both. The attributes and modifiers apply to all of the members declared by the *field-declaration*. It is an error for the same modifier to appear multiple times in a field declaration.

The *type* of a *field-declaration* specifies the type of the members introduced by the declaration. The type is followed by a list of *variable-declarator*s, each of which introduces a new

www.it-ebooks.info

member. A *variable-declarator* consists of an *identifier* that names that member, optionally followed by an "=" token and a *variable-initializer* (§10.5.5) that gives the initial value of that member.

The *type* of a field must be at least as accessible as the field itself (§3.5.4).

The value of a field is obtained in an expression using a *simple-name* (§7.6.2) or a *member-access* (§7.6.4). The value of a non-read-only field is modified using an *assignment* (§7.17). The value of a non-read-only field can be both obtained and modified using postfix increment and decrement operators (§7.6.9) and prefix increment and decrement operators (§7.7.5).

A field declaration that declares multiple fields is equivalent to multiple declarations of single fields with the same attributes, modifiers, and type. For example,

```
class A
{
    public static int X = 1, Y, Z = 100;
}
```

is equivalent to

```
class A
{
    public static int X = 1;
    public static int Y;
    public static int Z = 100;
}
```

10.5.1 Static and Instance Fields

When a field declaration includes a `static` modifier, the fields introduced by the declaration are ***static fields***. When no `static` modifier is present, the fields introduced by the declaration are *instance fields*. Static fields and instance fields are two of the several kinds of variables (§5) supported by C#, and at times they are referred to as ***static variables*** and ***instance variables***, respectively.

A static field is not part of a specific instance; instead, it is shared among all instances of a closed type (§4.4.2). No matter how many instances of a closed class type are created, there is only ever one copy of a static field for the associated application domain.

For example:

```
class C<V>
{
    static int count = 0;

    public C() {
        count++;
    }
```


www.it-ebooks.info

```
    public static int Count {
        get { return count; }
    }
}


class Application
{
    static void Main() {
        C<int> x1 = new C<int>();
        Console.WriteLine(C<int>.Count);          // Prints 1

        C<double> x2 = new C<double>();
        Console.WriteLine(C<int>.Count);          // Prints 1

        C<int> x3 = new C<int>();
        Console.WriteLine(C<int>.Count);          // Prints 2
    }
}
```

An instance field belongs to an instance. Specifically, every instance of a class contains a separate set of all the instance fields of that class.

When a field is referenced in a *member-access* (§7.6.4) of the form `E.M`, if `M` is a static field, `E` must denote a type containing `M`; if `M` is an instance field, E must denote an instance of a type containing `M`.

The differences between static and instance members are discussed further in §10.3.7.

10.5.2 Read-only Fields

When a *field-declaration* includes a `readonly` modifier, the fields introduced by the declaration are ***read-only fields***. Direct assignments to `readonly` fields can occur only as part of that declaration or in an instance constructor or static constructor in the same class. (A `readonly` field can be assigned to multiple times in these contexts.) Specifically, direct assignments to a `readonly` field are permitted only in the following contexts:

- In the *variable-declarator* that introduces the field (by including a *variable-initializer* in the declaration).
- For an instance field, in the instance constructors of the class that contains the field declaration; for a static field, in the static constructor of the class that contains the field declaration. These are also the only contexts in which it is valid to pass a `readonly` field as an `out` or `ref` parameter.

Attempting to assign to a `readonly` field or pass it as an `out` or `ref` parameter in any other context is a compile-time error.

www.it-ebooks.info

VLADIMIR RESHETNIKOV Only instance `readonly` fields of the `this` object can be assigned in instance constructors, but not fields of any other objects of the same type:

```
class A
{
    readonly int x;

    A(A a)
    {
        this.x = 1;     // Okay
        a.x = 1;        // Error CS0191: A read-only field
                        // cannot be assigned to
    }
}
```

Also, a `readonly` field cannot be assigned to (or passed as a `ref` or `out` parameter) in an anonymous function, even if the function is located within a constructor.

10.5.2.1 *Using Static Read-only Fields for Constants*

A `static readonly` field is useful when a symbolic name for a constant value is desired, but when the type of the value is not permitted in a `const` declaration, or when the value cannot be computed at compile time. In the example

```
public class Color
{
    public static readonly Color Black = new Color(0, 0, 0);
    public static readonly Color White = new Color(255, 255, 255);
    public static readonly Color Red = new Color(255, 0, 0);
    public static readonly Color Green = new Color(0, 255, 0);
    public static readonly Color Blue = new Color(0, 0, 255);

    private byte red, green, blue;

    public Color(byte r, byte g, byte b) {
        red = r;
        green = g;
        blue = b;
    }
}
```

the `Black`, `White`, `Red`, `Green`, and `Blue` members cannot be declared as `const` members because their values cannot be computed at compile time. Declaring them as `static readonly` instead has much the same effect.

10.5.2.2 *Versioning of Constants and Static Read-only Fields*

Constants and `readonly` fields have different binary versioning semantics. When an expression references a constant, the value of the constant is obtained at compile time, but when

www.it-ebooks.info

an expression references a `readonly` field, the value of the field is not obtained until run-time. Consider an application that consists of two separate programs:

```
using System;

namespace Program1
{
    public class Utils
    {
        public static readonly int X = 1;
    }
}

namespace Program2
{
    class Test
    {
        static void Main() {
            Console.WriteLine(Program1.Utils.X);
        }
    }
}
```

The `Program1` and `Program2` namespaces denote two programs that are compiled separately. Because `Program1.Utils.X` is declared as a `static readonly` field, the value output by the `Console.WriteLine` statement is not known at compile time, but rather is obtained at runtime. Thus, if the value of `X` is changed and `Program1` is recompiled, the `Console.WriteLine` statement will output the new value even if `Program2` isn't recompiled. However, had `X` been a constant, the value of `X` would have been obtained at the time `Program2` was compiled, and would remain unaffected by changes in `Program1` until `Program2` is recompiled.

BILL WAGNER This discussion justifies why `readonly` should often be preferred to `const`.

JON SKEET While I largely agree with Bill that `readonly` should usually be preferred, there is a benefit to `const` in some situations. Expressions that are built up of constants can be evaluated at compile time, rather than being recomputed on every access. For cases computing a string constant, this practice can avoid new strings being created each time the expression is evaluated, too. Some numbers really *are* natural constants: the number of milliseconds in a second, or the minimum value of an `int`, for example. When there is absolutely no chance that a value will ever change, `const` makes sense. When there is any doubt at all, `readonly` is safer.

www.it-ebooks.info

10.5.3 Volatile Fields

When a *field-declaration* includes a `volatile` modifier, the fields introduced by that declaration are ***volatile fields***.

For non-volatile fields, optimization techniques that reorder instructions can lead to unexpected and unpredictable results in multithreaded programs that access fields without synchronization, such as that provided by the *lock-statement* (§8.12). These optimizations can be performed by the compiler, by the runtime system, or by hardware. For volatile fields, such reordering optimizations are restricted:

- A read of a volatile field is called a ***volatile read***. A volatile read has "acquire semantics"; that is, it is guaranteed to occur prior to any references to memory that occur after it in the instruction sequence.
- A write of a volatile field is called a ***volatile write***. A volatile write has "release semantics"; that is, it is guaranteed to happen after any memory references prior to the write instruction in the instruction sequence.

These restrictions ensure that all threads will observe volatile writes performed by any other thread in the order in which they were performed. A conforming implementation is not required to provide a single total ordering of volatile writes as seen from all threads of execution. The type of a volatile field must be one of the following:

- A *reference-type*.
- The type `byte`, `sbyte`, `short`, `ushort`, `int`, `uint`, `char`, `float`, `bool`, `System.IntPtr`, or `System.UIntPtr`.
- An *enum-type* having an enum base type of `byte`, `sbyte`, `short`, `ushort`, `int`, or `uint`.

The example

```
using System;
using System.Threading;

class Test
{
    public static int result;
    public static volatile bool finished;

    static void Thread2() {
        result = 143;
        finished = true;
    }

    static void Main() {
        finished = false;

        // Run Thread2() in a new thread
        new Thread(new ThreadStart(Thread2)).Start();
```


www.it-ebooks.info

```
            // Wait for Thread2 to signal that it has a
            // result by setting finished to true
            for (;;) {
                if (finished) {
                    Console.WriteLine("result = {0}", result);
                    return;
                }
            }
        }
    }
```

produces the following output:

```
result = 143
```

In this example, the method `Main` starts a new thread that runs the method `Thread2`. This method stores a value into a non-volatile field called `result`, then stores `true` into the volatile field `finished`. The main thread waits for the field `finished` to be set to `true`, then reads the field `result`. Since `finished` has been declared `volatile`, the main thread must read the value `143` from the field `result`. If the field `finished` had not been declared as `volatile`, then it would be permissible for the store to `result` to be visible to the main thread *after* the store to `finished`, and hence for the main thread to read the value `0` from the field `result`. Declaring `finished` as a `volatile` field prevents any such inconsistency.

JOSEPH ALBAHARI Fields that are always accessed within a `lock` statement (§8.12) do not need to be declared with the `volatile` keyword. As a consequence, the runtime must ensure that any ordering optimization of fields used between `Monitor.Enter` and `Monitor.Exit` does not extend outside the scope of these statements.

JON SKEET Lock-free concurrent code is hard to get right. The precise guarantees of volatility are hard to reason about, making it far from obvious that code is truly correct when reading it. I tend to leave the details of low-level lock-free code to the experts, instead preferring to build on higher-level libraries written by experts. It's still important that the specification details the behavior for those experts' benefit, of course.

10.5.4 Field Initialization

The initial value of a field, whether it be a static field or an instance field, is the default value (§5.2) of the field's type. It is not possible to observe the value of a field before

www.it-ebooks.info

this default initialization has occurred; thus a field is never "uninitialized." The example

```
using System;

class Test
{
    static bool b;
    int i;

    static void Main() {
        Test t = new Test();
        Console.WriteLine("b = {0}, i = {1}", b, t.i);
    }
}
```

produces the output

```
b = False, i = 0
```

because `b` and `i` are both automatically initialized to default values.

10.5.5 Variable Initializers

Field declarations may include *variable-initializer*s. For static fields, variable initializers correspond to assignment statements that are executed during class initialization. For instance fields, variable initializers correspond to assignment statements that are executed when an instance of the class is created.

The example

```
using System;

class Test
{
    static double x = Math.Sqrt(2.0);
    int i = 100;
    string s = "Hello";

    static void Main() {
        Test a = new Test();
        Console.WriteLine("x = {0}, i = {1}, s = {2}", x, a.i, a.s);
    }
}
```

produces the output

```
x = 1.4142135623731, i = 100, s = Hello
```

because an assignment to `x` occurs when static field initializers execute, and assignments to `i` and `s` occur when the instance field initializers execute.

www.it-ebooks.info

The default value initialization described in §10.5.4 occurs for all fields, including fields that have variable initializers. Thus, when a class is initialized, all static fields in that class are first initialized to their default values, and then the static field initializers are executed in textual order. Likewise, when an instance of a class is created, all instance fields in that instance are first initialized to their default values, and then the instance field initializers are executed in textual order.

It is possible for static fields with variable initializers to be observed in their default value state, but this is strongly discouraged as a matter of style. The example

```
using System;

class Test
{
    static int a = b + 1;
    static int b = a + 1;

    static void Main() {
        Console.WriteLine("a = {0}, b = {1}", a, b);
    }
}
```

exhibits this behavior. Despite the circular definitions of `a` and `b`, the program is valid. It results in the output

```
a = 1, b = 2
```

because the static fields `a` and `b` are initialized to `0` (the default value for `int`) before their initializers are executed. When the initializer for `a` runs, the value of `b` is zero, so `a` is initialized to `1`. When the initializer for `b` runs, the value of `a` is already `1`, so `b` is initialized to `2`.

> **BILL WAGNER** This case is easier to understand when static variables are declared in partial classes in multiple source units. Furthermore, because you don't know in which order the source units will be included, you cannot know which variable will be initialized first.

10.5.5.1 *Static Field Initialization*

The static field variable initializers of a class correspond to a sequence of assignments that are executed in the textual order in which they appear in the class declaration. If a static constructor (§10.12) exists in the class, execution of the static field initializers occurs immediately prior to executing that static constructor. Otherwise, the static field initializers are

www.it-ebooks.info

executed at an implementation-dependent time prior to the first use of a static field of that class. The example

```
using System;

class Test
{
    static void Main() {
        Console.WriteLine("{0} {1}", B.Y, A.X);
    }

    public static int F(string s) {
        Console.WriteLine(s);
        return 1;
    }
}

class A
{
    public static int X = Test.F("Init A");
}

class B
{
    public static int Y = Test.F("Init B");
}
```

might produce either this output:

```
Init A
Init B
1 1
```

or this output:

```
Init B
Init A
1 1
```

The variation is possible because the execution of `X`'s initializer and `Y`'s initializer could occur in either order; they are constrained only to occur before the references to those fields. However, in the example

```
using System;

class Test
{
    static void Main() {
        Console.WriteLine("{0} {1}", B.Y, A.X);
    }

    public static int F(string s) {
        Console.WriteLine(s);
```


www.it-ebooks.info

```
        return 1;
    }
}

class A
{
    static A() {}

    public static int X = Test.F("Init A");
}

class B
{
    static B() {}

    public static int Y = Test.F("Init B");
}
```

the output must be

```
Init B
Init A
1 1
```

because the rules for when static constructors execute (as defined in §10.12) provide that `B`'s static constructor (and hence `B`'s static field initializers) must run before `A`'s static constructor and field initializers.

10.5.5.2 *Instance Field Initialization*

The instance field variable initializers of a class correspond to a sequence of assignments that are executed immediately upon entry to any one of the instance constructors (§10.11.1) of that class. The variable initializers are executed in the textual order in which they appear in the class declaration. The class instance creation and initialization process is described further in §10.11.

A variable initializer for an instance field cannot reference the instance being created. Thus it is a compile-time error to reference `this` in a variable initializer, because it is a compile-time error for a variable initializer to reference any instance member through a *simple-name*. In the example

```
class A
{
    int x = 1;
    int y = x + 1; // Error: reference to instance member of this
}
```

the variable initializer for `y` results in a compile-time error because it references a member of the instance being created.

www.it-ebooks.info

10.6 Methods

A ***method*** is a member that implements a computation or action that can be performed by an object or class. Methods are declared using *method-declaration*s:

method-declaration:
    method-header *method-body*

method-header:
    attributes$_{opt}$ *method-modifiers*$_{opt}$ `partial`$_{opt}$ *return-type* *member-name*
        type-parameter-list$_{opt}$ `(` *formal-parameter-list*$_{opt}$ `)`
        type-parameter-constraints-clauses$_{opt}$

method-modifiers:
    method-modifier
    method-modifiers *method-modifier*

method-modifier:
    `new`
    `public`
    `protected`
    `internal`
    `private`
    `static`
    `virtual`
    `sealed`
    `override`
    `abstract`
    `extern`

return-type:
    type
    `void`

member-name:
    identifier
    interface-type `.` *identifier*

method-body:
    block
    `;`

www.it-ebooks.info

A *method-declaration* may include a set of *attributes* (§17) and a valid combination of the four access modifiers (§10.3.5) and the `new` (§10.3.4), `static` (§10.6.2), `virtual` (§10.6.3), `override` (§10.6.4), `sealed` (§10.6.5), `abstract` (§10.6.6), and `extern` (§10.6.7) modifiers.

A declaration has a valid combination of modifiers if all of the following are true:

- The declaration includes a valid combination of access modifiers (§10.3.5).
- The declaration does not include the same modifier multiple times.
- The declaration includes at most one of the following modifiers: `static`, `virtual`, and `override`.
- The declaration includes at most one of the following modifiers: `new` and `override`.
- If the declaration includes the `abstract` modifier, then the declaration does not include any of the following modifiers: `static`, `virtual`, `sealed`, or `extern`.
- If the declaration includes the `private` modifier, then the declaration does not include any of the following modifiers: `virtual`, `override`, or `abstract`.
- If the declaration includes the `sealed` modifier, then the declaration also includes the `override` modifier.
- If the declaration includes the `partial` modifier, then it does not include any of the following modifiers: `new`, `public`, `protected`, `internal`, `private`, `virtual`, `sealed`, `override`, `abstract`, or `extern`.

The *return-type* of a method declaration specifies the type of the value computed and returned by the method. The *return-type* is `void` if the method does not return a value. If the declaration includes the `partial` modifier, then the return type must be `void`.

The *member-name* specifies the name of the method. Unless the method is an explicit interface member implementation (§13.4.1), the *member-name* is simply an *identifier*. For an explicit interface member implementation, the *member-name* consists of an *interface-type* followed by a "`.`" and an *identifier*.

The optional *type-parameter-list* specifies the type parameters of the method (§10.1.3). If a *type-parameter-list* is specified, the method is a ***generic method***. If the method has an `extern` modifier, a *type-parameter-list* cannot be specified.

The optional *formal-parameter-list* specifies the parameters of the method (§10.6.1).

The optional *type-parameter-constraints-clauses* specify constraints on individual type parameters (§10.1.5) and may be specified only if a *type-parameter-list* is also supplied, and if the method does not have an `override` modifier.

www.it-ebooks.info

The *return-type* and each of the types referenced in the *formal-parameter-list* of a method must be at least as accessible as the method itself (§3.5.4).

For `abstract` and `extern` methods, the *method-body* consists simply of a semicolon. For `partial` methods, the *method-body* may consist of either a semicolon or a *block*. For all other methods, the *method-body* consists of a *block*, which specifies the statements to execute when the method is invoked.

The name, the type parameter list, and the formal parameter list of a method define the signature (§3.6) of the method. Specifically, the signature of a method consists of its name, the number of type parameters, and the number, modifiers, and types of its formal parameters. For these purposes, any type parameter of the method that occurs in the type of a formal parameter is identified not by its name, but rather by its ordinal position in the type argument list of the method. The return type is not part of a method's signature, nor are the names of the type parameters or the formal parameters.

The name of a method must differ from the names of all other nonmethods declared in the same class. In addition, the signature of a method must differ from the signatures of all other methods declared in the same class, and two methods declared in the same class may not have signatures that differ solely by `ref` and `out`.

The method's *type-parameter*s are in scope throughout the *method-declaration*, and can be used to form types throughout that scope in *return-type*, *method-body*, and *type-parameter-constraints-clauses* but not in *attributes*.

All formal parameters and type parameters must have different names.

10.6.1 Method Parameters

The parameters of a method, if any, are declared by the method's *formal-parameter-list*.

formal-parameter-list:
    fixed-parameters
    fixed-parameters **,** *parameter-array*
    parameter-array

fixed-parameters:
    fixed-parameter
    fixed-parameters **,** *fixed-parameter*

fixed-parameter:
    $attributes_{opt}$ $parameter\text{-}modifier_{opt}$ *type* *identifier* $default\text{-}argument_{opt}$

default-argument:
    = *expression*

www.it-ebooks.info

parameter-modifier:
    `ref`
    `out`
    `this`

parameter-array:
    attributes$_{opt}$ `params` *array-type* *identifier*

The formal parameter list consists of one or more comma-separated parameters, of which only the last may be a *parameter-array*.

A *fixed-parameter* consists of an optional set of *attributes* (§17); an optional `ref`, `out`, or `this` modifier; a *type*; an *identifier*; and an optional *default-argument*. Each *fixed-parameter* declares a parameter of the given type with the given name. The `this` modifier designates the method as an extension method and is allowed only on the first parameter of a static method. Extension methods are further described in §10.6.9.

A *fixed-parameter* with a *default-argument* is known as an ***optional parameter***, whereas a *fixed-parameter* without a *default-argument* is a ***required parameter***. A required parameter may not appear after an optional parameter in a *formal-parameter-list*.

> **CHRISTIAN NAGEL** Attention must be paid to optional parameters in regard to versioning. The compiler takes default arguments and adds them to the caller in the assembly of the caller. If the default argument changes to a new value with a new version of the code, the caller still contains the old value if the code is not recompiled.

A `ref` or `out` parameter cannot have a *default-argument*. The *expression* in a *default-argument* must be one of the following:

- A *constant-expression*.
- An expression of the form `new S()`, where `S` is a value type.
- An expression of the form `default(S)`, where `S` is a value type.

The *expression* must be implicitly convertible by an identity or nullable conversion to the type of the parameter.

If optional parameters occur in an implementing partial method declaration (§10.2.7), in an explicit interface member implementation (§13.4.1), or in a single-parameter indexer declaration (§10.9), the compiler should give a warning, since these members can never be invoked in a way that permits arguments to be omitted.

www.it-ebooks.info

A *parameter-array* consists of an optional set of *attributes* (§17), a `params` modifier, an *array-type*, and an *identifier*. A parameter array declares a single parameter of the given array type with the given name. The *array-type* of a parameter array must be a single-dimensional array type (§12.1). In a method invocation, a parameter array permits either a single argument of the given array type or zero or more arguments of the array element type to be specified. Parameter arrays are described further in §10.6.1.4.

A *parameter-array* may occur after an optional parameter, but cannot have a default value—the omission of arguments for a *parameter-array* would instead result in the creation of an empty array.

The following example illustrates different kinds of parameters:

```
public void M(
    ref int    i,
    decimal    d,
    bool       b = false,
    bool?      n = false,
    string     s = "Hello",
    object     o = null,
    T          t = default(T),
    params int[] a
) { }
```

In the *formal-parameter-list* for `M`, `i` is a required `ref` parameter; `d` is a required value parameter; `b`, `s`, `o`, and `t` are optional value parameters; and `a` is a parameter array.

A method declaration creates a separate declaration space for parameters, type parameters, and local variables. Names are introduced into this declaration space by the type parameter list and the formal parameter list of the method and by local variable declarations in the *block* of the method. It is an error for two members of a method declaration space to have the same name. It is an error for the method declaration space and the local variable declaration space of a nested declaration space to contain elements with the same name.

A method invocation (§7.6.5.1) creates a copy, specific to that invocation, of the formal parameters and local variables of the method, and the argument list of the invocation assigns values or variable references to the newly created formal parameters. Within the *block* of a method, formal parameters can be referenced by their identifiers in *simple-name* expressions (§7.6.2).

There are four kinds of formal parameters:

- Value parameters, which are declared without any modifiers.
- Reference parameters, which are declared with the `ref` modifier.
- Output parameters, which are declared with the `out` modifier.
- Parameter arrays, which are declared with the `params` modifier.

www.it-ebooks.info

As described in §3.6, the `ref` and `out` modifiers are part of a method's signature, but the `params` modifier is not.

10.6.1.1 *Value Parameters*

A parameter declared with no modifiers is a value parameter. A value parameter corresponds to a local variable that gets its initial value from the corresponding argument supplied in the method invocation.

When a formal parameter is a value parameter, the corresponding argument in a method invocation must be an expression that is implicitly convertible (§6.1) to the formal parameter type.

A method is permitted to assign new values to a value parameter. Such assignments affect only the local storage location represented by the value parameter; they have no effect on the actual argument given in the method invocation.

10.6.1.2 *Reference Parameters*

A parameter declared with a `ref` modifier is a reference parameter. Unlike a value parameter, a reference parameter does not create a new storage location. Instead, a reference parameter represents the same storage location as the variable given as the argument in the method invocation.

When a formal parameter is a reference parameter, the corresponding argument in a method invocation must consist of the keyword `ref` followed by a *variable-reference* (§5.3.3) of the same type as the formal parameter. A variable must be definitely assigned before it can be passed as a reference parameter.

Within a method, a reference parameter is always considered definitely assigned.

A method declared as an iterator (§10.14) cannot have reference parameters.

The example

```
using System;

class Test
{
    static void Swap(ref int x, ref int y) {
        int temp = x;
        x = y;
        y = temp;
    }

    static void Main() {
        int i = 1, j = 2;
        Swap(ref i, ref j);
        Console.WriteLine("i = {0}, j = {1}", i, j);
    }
}
```


www.it-ebooks.info

produces the following output:

```
i = 2, j = 1
```

For the invocation of `Swap` in `Main`, `x` represents `i` and `y` represents `j`. Thus the invocation has the effect of swapping the values of `i` and `j`.

In a method that takes reference parameters, it is possible for multiple names to represent the same storage location. In the example

```
class A
{
    string s;

    void F(ref string a, ref string b) {
        s = "One";
        a = "Two";
        b = "Three";
    }

    void G() {
        F(ref s, ref s);
    }
}
```

the invocation of `F` in `G` passes a reference to `s` for both `a` and `b`. Thus, for that invocation, the names `s`, `a`, and `b` all refer to the same storage location, and the three assignments all modify the instance field `s`.

10.6.1.3 *Output Parameters*

A parameter declared with an `out` modifier is an output parameter. Similar to a reference parameter, an output parameter does not create a new storage location. Instead, an output parameter represents the same storage location as the variable given as the argument in the method invocation.

When a formal parameter is an output parameter, the corresponding argument in a method invocation must consist of the keyword `out` followed by a *variable-reference* (§5.3.3) of the same type as the formal parameter. A variable need not be definitely assigned before it can be passed as an output parameter, but following an invocation where a variable was passed as an output parameter, the variable is considered definitely assigned.

Within a method, just like a local variable, an output parameter is initially considered unassigned and must be definitely assigned before its value is used.

Every output parameter of a method must be definitely assigned before the method returns.

www.it-ebooks.info

A method declared as a partial method (§10.2.7) or an iterator (§10.14) cannot have output parameters.

Output parameters are typically used in methods that produce multiple return values.

> **BILL WAGNER** Of course, you could create a `struct` or `class` to return the multiple values, and that would obviate the need for output parameters. In addition, the new `Tuple<>` generic classes can be used to return multiple values.

> **JON SKEET** One example of Bill's point could have been `int.TryParse` in the .NET Framework. It effectively returns two values: the parsed integer and a boolean flag to indicate whether the operation was successful. Nullable value types in C# already provide exactly this combination, so the current method signature of this:
>
> ```
> bool TryParse(string s, out int result)
> ```
>
> could have instead been this:
>
> ```
> int? TryParse(string s)
> ```
>
> The same is true of similar calls such as `decimal.TryParse` and so on.

For example:

```
using System;

class Test
{
    static void SplitPath(string path, out string dir, out string name) {
        int i = path.Length;
        while (i > 0) {
            char ch = path[i - 1];
            if (ch == '\\' || ch == '/' || ch == ':') break;
            i--;
        }
        dir = path.Substring(0, i);
        name = path.Substring(i);
    }

    static void Main() {
        string dir, name;
        SplitPath("c:\\Windows\\System\\hello.txt", out dir, out name);
        Console.WriteLine(dir);
        Console.WriteLine(name);
    }
}
```


www.it-ebooks.info

The example produces the following output:

```
c:\Windows\System\
hello.txt
```

Note that the `dir` and `name` variables can be unassigned before they are passed to `SplitPath`, and that they are considered definitely assigned following the call.

10.6.1.4 *Parameter Arrays*

A parameter declared with a `params` modifier is a parameter array. If a formal parameter list includes a parameter array, it must be the last parameter in the list and it must be of a single-dimensional array type. For example, the types `string[]` and `string[][]` can be used as the type of a parameter array, but the type `string[,]` cannot. It is not possible to combine the `params` modifier with the modifiers `ref` and `out`.

A parameter array permits arguments to be specified in one of two ways in a method invocation:

- The argument given for a parameter array can be a single expression that is implicitly convertible (§6.1) to the parameter array type. In this case, the parameter array acts precisely like a value parameter.
- Alternatively, the invocation can specify zero or more arguments for the parameter array, where each argument is an expression that is implicitly convertible (§6.1) to the element type of the parameter array. In this case, the invocation creates an instance of the parameter array type with a length corresponding to the number of arguments, initializes the elements of the array instance with the given argument values, and uses the newly created array instance as the actual argument.

Except for allowing a variable number of arguments in an invocation, a parameter array is precisely equivalent to a value parameter (§10.6.1.1) of the same type.

The example

```
using System;

class Test
{
    static void F(params int[] args) {
        Console.Write("Array contains {0} elements:", args.Length);
        foreach (int i in args)
            Console.Write(" {0}", i);
        Console.WriteLine();
    }
```


www.it-ebooks.info

```
        static void Main() {
            int[] arr = {1, 2, 3};
            F(arr);
            F(10, 20, 30, 40);
            F();
        }
    }
```

produces the following output:

```
Array contains 3 elements: 1 2 3
Array contains 4 elements: 10 20 30 40
Array contains 0 elements:
```

The first invocation of `F` simply passes the array `a` as a value parameter. The second invocation of `F` automatically creates a four-element `int[]` with the given element values and passes that array instance as a value parameter. Likewise, the third invocation of `F` creates a zero-element `int[]` and passes that instance as a value parameter. The second and third invocations are precisely equivalent to writing

```
F(new int[] {10, 20, 30, 40});
F(new int[] {});
```

When performing overload resolution, a method with a parameter array may be applicable either in its normal form or in its expanded form (§7.5.3.1). The expanded form of a method is available only if the normal form of the method is not applicable and only if a method with the same signature as the expanded form is not already declared in the same type.

The example

```
using System;

class Test
{
    static void F(params object[] a) {
        Console.WriteLine("F(object[])");
    }

    static void F() {
        Console.WriteLine("F()");
    }

    static void F(object a0, object a1) {
        Console.WriteLine("F(object,object)");
    }

    static void Main() {
        F();
        F(1);
        F(1, 2);
        F(1, 2, 3);
        F(1, 2, 3, 4);
    }
}
```


www.it-ebooks.info

produces the following output:

```
F();
F(object[]);
F(object,object);
F(object[]);
F(object[]);
```

> **BILL WAGNER** The more methods you create that could possibly be suitable methods, the more trouble you create for your users. More compiler ambiguity creates more ambiguity for your client developers as well.

In the example, two of the possible expanded forms of the method with a parameter array are already included in the class as regular methods. These expanded forms are, therefore, not considered when performing overload resolution, and the first and third method invocations select the regular methods. When a class declares a method with a parameter array, it is not uncommon to also include some of the expanded forms as regular methods. By doing so, it is possible to avoid the allocation of an array instance that occurs when an expanded form of a method with a parameter array is invoked.

When the type of a parameter array is `object[]`, a potential ambiguity arises between the normal form of the method and the expended form for a single `object` parameter. The reason for the ambiguity is that an `object[]` is itself implicitly convertible to type `object`. The ambiguity presents no problem, however, since it can be resolved by inserting a cast if needed.

The example

```
using System;

class Test
{
    static void F(params object[] args) {
        foreach (object o in args) {
            Console.Write(o.GetType().FullName);
            Console.Write(" ");
        }
        Console.WriteLine();
    }

    static void Main() {
        object[] a = {1, "Hello", 123.456};
        object o = a;
```


www.it-ebooks.info

```
            F(a);
            F((object)a);
            F(o);
            F((object[])o);
        }
    }
```

produces the following output:

```
System.Int32 System.String System.Double
System.Object[]
System.Object[]
System.Int32 System.String System.Double
```

In the first and last invocations of `F`, the normal form of `F` is applicable because an implicit conversion exists from the argument type to the parameter type (both are of type `object[]`). Thus overload resolution selects the normal form of `F`, and the argument is passed as a regular value parameter. In the second and third invocations, the normal form of `F` is not applicable because no implicit conversion exists from the argument type to the parameter type (type `object` cannot be implicitly converted to type `object[]`). However, the expanded form of `F` is applicable, so it is selected by overload resolution. As a result, a one-element `object[]` is created by the invocation, and the single element of the array is initialized with the given argument value (which itself is a reference to an `object[]`).

10.6.2 Static and Instance Methods

When a method declaration includes a `static` modifier, that method is said to be a static method. When no `static` modifier is present, the method is said to be an instance method.

A static method does not operate on a specific instance, and it is a compile-time error to refer to `this` in a static method.

An instance method operates on a given instance of a class, and that instance can be accessed as `this` (§7.6.7).

When a method is referenced in a *member-access* (§7.6.4) of the form `E.M`, if `M` is a static method, `E` must denote a type containing `M`; if `M` is an instance method, `E` must denote an instance of a type containing `M`.

The differences between static and instance members are discussed further in §10.3.7.

www.it-ebooks.info

10.6.3 Virtual Methods

When an instance method declaration includes a `virtual` modifier, that method is said to be a virtual method. When no `virtual` modifier is present, the method is said to be a nonvirtual method.

> **BILL WAGNER** You can create virtual generic methods, even in a nongeneric class. When you do so, any overrides must also be generic. You cannot override a particular instantiation.

> **JON SKEET** Private methods cannot be virtual, even though there is one corner case where it would make sense: A nested class derived from its own container class could override it (with another private method). While it's a tiny bit inelegant for this possibility to be forbidden by the specification, it means that methods that are *accidentally* private and virtual can be treated as errors—and this is a far more common case.

The implementation of a nonvirtual method is invariant: The implementation is the same whether the method is invoked on an instance of the class in which it is declared or an instance of a derived class. In contrast, the implementation of a virtual method can be superseded by derived classes. The process of superseding the implementation of an inherited virtual method is known as ***overriding*** that method (§10.6.4).

In a virtual method invocation, the ***runtime type*** of the instance for which that invocation takes place determines the actual method implementation to invoke. In a nonvirtual method invocation, the ***compile-time type*** of the instance is the determining factor. In precise terms, when a method named `N` is invoked with an argument list `A` on an instance with a compile-time type `C` and a runtime type `R` (where `R` is either `C` or a class derived from `C`), the invocation is processed as follows:

- First, overload resolution is applied to `C`, `N`, and `A` to select a specific method `M` from the set of methods declared in and inherited by `C`. This is described in §7.6.5.1.
- Then, if `M` is a nonvirtual method, `M` is invoked.
- Otherwise, `M` is a virtual method, and the most derived implementation of `M` with respect to `R` is invoked.

For every virtual method declared in or inherited by a class, there exists a ***most derived implementation*** of the method with respect to that class. The most derived implementation of a virtual method `M` with respect to a class `R` is determined as follows:

www.it-ebooks.info

- If R contains the introducing virtual declaration of M, then it is the most derived implementation of M.
- Otherwise, if R contains an override of M, then it is the most derived implementation of M.
- Otherwise, the most derived implementation of M with respect to R is the same as the most derived implementation of M with respect to the direct base class of R.

The following example illustrates the differences between virtual and nonvirtual methods:

```
using System;

class A
{
    public void F() { Console.WriteLine("A.F"); }

    public virtual void G() { Console.WriteLine("A.G"); }
}

class B: A
{
    new public void F() { Console.WriteLine("B.F"); }

    public override void G() { Console.WriteLine("B.G"); }
}

class Test
{
    static void Main() {
        B b = new B();
        A a = b;
        a.F();
        b.F();
        a.G();
        b.G();
    }
}
```

In the example, A introduces a nonvirtual method F and a virtual method G. The class B introduces a *new* non-virtual method F, thus *hiding* the inherited F, and also *overrides* the inherited method G. The example produces the following output:

```
A.F
B.F
B.G
B.G
```

Notice that the statement a.G() invokes B.G, not A.G. This outcome occurs because the runtime type of the instance (which is B), not the compile-time type of the instance (which is A), determines the actual method implementation to invoke.

www.it-ebooks.info

Because methods are allowed to hide inherited methods, it is possible for a class to contain several virtual methods with the same signature. This does not present an ambiguity problem, since all but the most derived method are hidden. In the example

```
using System;

class A
{
    public virtual void F() { Console.WriteLine("A.F"); }
}

class B: A
{
    public override void F() { Console.WriteLine("B.F"); }
}

class C: B
{
    new public virtual void F() { Console.WriteLine("C.F"); }
}

class D: C
{
    public override void F() { Console.WriteLine("D.F"); }
}

class Test
{
    static void Main() {
        D d = new D();
        A a = d;
        B b = d;
        C c = d;
        a.F();
        b.F();
        c.F();
        d.F();
    }
}
```

the `C` and `D` classes contain two virtual methods with the same signature: the one introduced by `A` and the one introduced by `C`. The method introduced by `C` hides the method inherited from `A`. Thus the override declaration in `D` overrides the method introduced by `C`, and it is not possible for `D` to override the method introduced by `A`. The example produces the following output:

```
B.F
B.F
D.F
D.F
```

Note that it is possible to invoke the hidden virtual method by accessing an instance of `D` through a less derived type in which the method is not hidden.

www.it-ebooks.info

10.6.4 Override Methods

When an instance method declaration includes an `override` modifier, the method is said to be an ***override method***. An override method overrides an inherited virtual method with the same signature. Whereas a virtual method declaration *introduces* a new method, an override method declaration *specializes* an existing inherited virtual method by providing a new implementation of that method.

The method overridden by an `override` declaration is known as the ***overridden base method***. For an override method `M` declared in a class `C`, the overridden base method is determined by examining each base class type of `C`, starting with the direct base class type of `C` and continuing with each successive direct base class type, until in a given base class type at least one accessible method is located that has the same signature as `M` after substitution of type arguments. For the purposes of locating the overridden base method, a method is considered accessible if it is `public`, if it is `protected`, if it is `protected internal`, or if it is `internal` and declared in the same program as `C`.

A compile-time error occurs unless all of the following are true for an override declaration:

- An overridden base method can be located as described above.
- There is exactly one such overridden base method. This restriction has effect only if the base class type is a constructed type where the substitution of type arguments makes the signature of two methods the same.
- The overridden base method is a virtual, abstract, or override method. In other words, the overridden base method cannot be static or non-virtual.
- The overridden base method is not a sealed method.
- The override method and the overridden base method have the same return type.
- The override declaration and the overridden base method have the same declared accessibility. In other words, an override declaration cannot change the accessibility of the virtual method. However, if the overridden base method is `protected internal` and it is declared in a different assembly than the assembly containing the override method, then the override method's declared accessibility must be `protected`.
- The override declaration does not specify type–parameter–constraints–clauses. Instead, the constraints are inherited from the overridden base method. Note that constraints that are type parameters in the overridden method may be replaced by type arguments in the inherited constraint. This can lead to constraints that are not legal when explicitly specified, such as value types or sealed types.

www.it-ebooks.info

The following example demonstrates how the overriding rules work for generic classes:

```
abstract class C<T>
{
    public virtual T F() {...}

    public virtual C<T> G() {...}

    public virtual void H(C<T> x) {...}
}

class D: C<string>
{
    public override string F() {...}        // Okay

    public override C<string> G() {...}     // Okay

    public override void H(C<T> x) {...}    // Error: should be C<string>
}

class E<T,U>: C<U>
{
    public override U F() {...}             // Okay

    public override C<U> G() {...}          // Okay

    public override void H(C<T> x) {...}    // Error: should be C<U>
}
```

An override declaration can access the overridden base method using a *base-access* (§7.6.8). In the example

```
class A
{
    int x;

    public virtual void PrintFields() {
        Console.WriteLine("x = {0}", x);
    }
}

class B: A
{
    int y;

    public override void PrintFields() {
        base.PrintFields();
        Console.WriteLine("y = {0}", y);
    }
}
```

the `base.PrintFields()` invocation in `B` invokes the `PrintFields` method declared in `A`. A *base-access* disables the virtual invocation mechanism and simply treats the base method as a non-virtual method. Had the invocation in `B` been written `((A)this).PrintFields()`, it

www.it-ebooks.info

would recursively invoke the `PrintFields` method declared in `B`, not the one declared in `A`, since `PrintFields` is virtual and the runtime type of `((A)this)` is `B`.

Only by including an `override` modifier can a method override another method. In all other cases, a method with the same signature as an inherited method simply hides the inherited method. In the example

```
class A
{
    public virtual void F() {}
}

class B: A
{
    public virtual void F() {}          // Warning: hiding inherited F()
}
```

the `F` method in `B` does not include an `override` modifier and, therefore, does not override the `F` method in `A`. Rather, the `F` method in `B` hides the method in `A`, and a warning is reported because the declaration does not include a `new` modifier.

In the example

```
class A
{
    public virtual void F() {}
}

class B: A
{
    new private void F() {}             // Hides A.F within body of B
}

class C: B
{
    public override void F() {}         // Okay: overrides A.F
}
```

the `F` method in `B` hides the virtual `F` method inherited from `A`. Since the new `F` in `B` has `private` access, its scope includes only the class body of `B` and does not extend to `C`. Therefore, the declaration of `F` in `C` is permitted to override the `F` inherited from `A`.

10.6.5 Sealed Methods

When an instance method declaration includes a `sealed` modifier, that method is said to be a ***sealed method***. If an instance method declaration includes the `sealed` modifier, it must also include the `override` modifier. Use of the `sealed` modifier prevents a derived class from further overriding the method.

www.it-ebooks.info

In the example

```
using System;

class A
{
    public virtual void F() {
        Console.WriteLine("A.F");
    }

    public virtual void G() {
        Console.WriteLine("A.G");
    }
}

class B: A
{
    sealed override public void F() {
        Console.WriteLine("B.F");
    }

    override public void G() {
        Console.WriteLine("B.G");
    }
}

class C: B
{
    override public void G() {
        Console.WriteLine("C.G");
    }
}
```

the class `B` provides two override methods: an `F` method that has the `sealed` modifier and a `G` method that does not. Class `B`'s use of the sealed `modifier` prevents class `C` from further overriding `F`.

CHRIS SELLS When I want to derive from a class, it drives me crazy if that class is sealed or if a method is sealed, because those restrictions limit how I can use the class.

When I want to build a base class, I want it to be as locked down as possible so that I can test the possible derivation scenarios I want to support. I don't want to be trapped into supporting something crazy that some customer has done and that I never intended.

As with all things in software design, the use of the `sealed` modifier involves a balance. I tend to avoid the use of the `sealed` keyword unless I have to use it. If customers get crazy, well, that's what keeps things interesting.

www.it-ebooks.info

10.6.6 Abstract Methods

When an instance method declaration includes an `abstract` modifier, that method is said to be an ***abstract method***. Although an abstract method is implicitly also a virtual method, it cannot have the modifier `virtual`.

An abstract method declaration introduces a new virtual method but does not provide an implementation of that method. Instead, nonabstract derived classes are required to provide their own implementation by overriding that method. Because an abstract method provides no actual implementation, the *method-body* of an abstract method simply consists of a semicolon.

Abstract method declarations are permitted only in abstract classes (§10.1.1.1).

In the example

```
public abstract class Shape
{
    public abstract void Paint(Graphics g, Rectangle r);
}

public class Ellipse: Shape
{
    public override void Paint(Graphics g, Rectangle r) {
        g.DrawEllipse(r);
    }
}

public class Box: Shape
{
    public override void Paint(Graphics g, Rectangle r) {
        g.DrawRect(r);
    }
}
```

the `Shape` class defines the abstract notion of a geometrical shape object that can paint itself. The `Paint` method is abstract because there is no meaningful default implementation. The `Ellipse` and `Box` classes are concrete `Shape` implementations. Because these classes are nonabstract, they are required to override the `Paint` method and provide an actual implementation.

It is a compile-time error for a *base-access* (§7.6.8) to reference an abstract method. In the example

```
abstract class A
{
    public abstract void F();
}
```


www.it-ebooks.info

```
class B: A
{
    public override void F() {
        base.F();                   // Error: base.F is abstract
    }
}
```

a compile-time error is reported for the `base.F()` invocation because it references an abstract method.

An abstract method declaration is permitted to override a virtual method. This allows an abstract class to force reimplementation of the method in derived classes, and makes the original implementation of the method unavailable. In the example

```
using System;

class A
{
    public virtual void F() {
        Console.WriteLine("A.F");
    }
}

abstract class B: A
{
    public abstract override void F();
}

class C: B
{
    public override void F() {
        Console.WriteLine("C.F");
    }
}
```

class `A` declares a virtual method, class `B` overrides this method with an abstract method, and class `C` overrides the abstract method to provide its own implementation.

> **MAREK SAFAR** There are no visibility restrictions on abstract methods except that they cannot be `private`. That access choice can cause trouble when someone accidentally declares an `internal abstract` method within a `public abstract` class; such a class, even though it is declared `public`, cannot be used outside the class assembly.

10.6.7 External Methods

When a method declaration includes an `extern` modifier, that method is said to be an ***external method***. External methods are implemented externally, typically using a language

www.it-ebooks.info

other than C#. Because an external method declaration provides no actual implementation, the *method-body* of an external method simply consists of a semicolon. An external method may not be generic.

The `extern` modifier is typically used in conjunction with the `DllImport` attribute (§17.5.1), allowing external methods to be implemented by Dynamic Link Libraries (DLLs). The execution environment may support other mechanisms whereby implementations of external methods can be provided.

When an external method includes a `DllImport` attribute, the method declaration must also include a `static` modifier. This example demonstrates the use of the `extern` modifier and the `DllImport` attribute:

```
using System.Text;
using System.Security.Permissions;
using System.Runtime.InteropServices;

class Path
{
    [DllImport("kernel32", SetLastError=true)]
    static extern bool CreateDirectory(string name, SecurityAttribute sa);

    [DllImport("kernel32", SetLastError=true)]
    static extern bool RemoveDirectory(string name);

    [DllImport("kernel32", SetLastError=true)]
    static extern int GetCurrentDirectory(int bufSize, StringBuilder buf);

    [DllImport("kernel32", SetLastError=true)]
    static extern bool SetCurrentDirectory(string name);
}
```

10.6.8 Partial Methods

When a method declaration includes a `partial` modifier, that method is said to be a ***partial method***. Partial methods can be declared only as members of partial types (§10.2) and are subject to a number of restrictions. Partial methods are further described in §10.2.7.

10.6.9 Extension Methods

When the first parameter of a method includes the `this` modifier, that method is said to be an ***extension method***. Extension methods can be declared only in nongeneric, non-nested static classes. The first parameter of an extension method can have no modifiers other than `this`, and the parameter type cannot be a pointer type.

www.it-ebooks.info

> **PETER SESTOFT** The parameter type also cannot be `dynamic` (although that would really just mean `object`). But it can be `T`, where `T` is a type parameter of the extension method, like so:
>
> ```
> public static void Foo<T>(this T x) { ... }
> ```
>
> This is potentially useful, especially if `T` has a type constraint and there are more parameters whose type involves `T`.

The following is an example of a static class that declares two extension methods:

```
using System;
public static class Extensions
{
    public static int ToInt32(this string s) {
        return Int32.Parse(s);
    }

    public static T[] Slice<T>(this T[] source, int index, int count) {
        if (index < 0 || count < 0 || source.Length - index < count)
            throw new ArgumentException();
        T[] result = new T[count];
        Array.Copy(source, index, result, 0, count);
        return result;
    }
}
```

An extension method is a regular static method. In addition, where its enclosing static class is in scope, an extension method can be invoked using instance method invocation syntax (§7.6.5.2), using the receiver expression as the first argument.

The following program uses the extension methods declared above:

```
static class Program
{
    static void Main() {
        string[] strings = { "1", "22", "333", "4444" };
        foreach (string s in strings.Slice(1, 2)) {
            Console.WriteLine(s.ToInt32());
        }
    }
}
```

The `Slice` method is available on the `string[]`, and the `ToInt32` method is available on `string`, because they have been declared as extension methods. The meaning of the program is the same as the following, using ordinary static method calls:

```
static class Program
{
    static void Main() {
```


www.it-ebooks.info

```
        string[] strings = { "1", "22", "333", "4444" };
        foreach (string s in Extensions.Slice(strings, 1, 2)) {
            Console.WriteLine(Extensions.ToInt32(s));
        }
    }
}
```

PETER SESTOFT The `Slice` extension method is generic: It works on any array of `T`, regardless of which type parameter `T` is. You can also define generic extension methods that involve type bounds (method `IsSorted` below) and extension methods that work on particular type instances of a generic type (method `ConcatWith`). The latter also shows that an extension method may, of course, have default arguments (§10.6.1):

```
public static bool IsSorted<T>(this IEnumerable<T> xs)
    where T : IComparable<T>
{ ... }

public static string ConcatWith(this IEnumerable<String> xs, string glue = ", ")
{ ... }
```

CHRIS SELLS On the one hand, extension methods are a great way to add methods that a type designer forgot, while still keeping the syntax as if the designer had your every whim in mind. On the other hand, these methods are easy to abuse. I saw an extension method on `object` in an experimental system once, and it became an ongoing joke in our group to see if we could top it for confusing design.

PETER SESTOFT Here's a shot at a most widely applicable extension method. Any attempt at calling a nonexistent method `toString`, with lowercase `t`, on any object, with any number of arguments of any type, will call this instead:

```
public static string toString(this object x, params object[] args) {
    return "Make that ToString(), mate!";
}
```

Extension methods are non-virtual and do not override anything; hence one cannot replace the `ToString` method (with uppercase `T`) of an existing type `YourType` by declaring an extension method. The following declaration is legal but has no effect. A call `o.ToString()` will pick up `Object.ToString()` or some method that overrides or hides it, not this one:

```
public static string ToString(this YourType x) { ... }
```


www.it-ebooks.info

10.6.10 Method Body

The *method-body* of a method declaration consists of either a *block* or a semicolon.

Abstract and external method declarations do not provide a method implementation, so their method bodies simply consist of a semicolon. For any other method, the method body is a block (§8.2) that contains the statements to execute when that method is invoked.

When the return type of a method is `void`, `return` statements (§8.9.4) in that method's body are not permitted to specify an expression. If execution of the method body of a void method completes normally (that is, if control flows off the end of the method body), that method simply returns to its caller.

When the return type of a method is not `void`, each `return` statement in that method's body must specify an expression that is implicitly convertible to the return type. The end point of the method body of a value-returning method must not be reachable. In other words, in a value-returning method, control is not permitted to flow off the end of the method body.

In the example

```
class A
{
    public int F() {}           // Error: return value required

    public int G() {
        return 1;
    }

    public int H(bool b) {
        if (b) {
            return 1;
        }
        else {
            return 0;
        }
    }
}
```

the value-returning `F` method results in a compile-time error because control can flow off the end of the method body. The `G` and `H` methods are correct because all possible execution paths end in a return statement that specifies a return value.

10.6.11 Method Overloading

The method overload resolution rules are described in §7.5.2.

www.it-ebooks.info

10.7 Properties

A ***property*** is a member that provides access to a characteristic of an object or a class. Examples of properties include the length of a string, the size of a font, the caption of a window, the name of a customer, and so on. Properties are a natural extension of fields—both are named members with associated types, and the syntax for accessing fields and properties is the same. However, unlike fields, properties do not denote storage locations. Instead, properties have ***accessors*** that specify the statements to be executed when their values are read or written. Properties thus provide a mechanism for associating actions with the reading and writing of an object's attributes; furthermore, they permit such attributes to be computed.

Properties are declared using *property-declaration*s:

property-declaration:
    attributes$_{opt}$ *property-modifiers*$_{opt}$ *type* *member-name* `{` *accessor-declarations* `}`

property-modifiers:
    property-modifier
    property-modifiers *property-modifier*

property-modifier:
    `new`
    `public`
    `protected`
    `internal`
    `private`
    `static`
    `virtual`
    `sealed`
    `override`
    `abstract`
    `extern`

member-name:
    identifier
    interface-type `.` *identifier*

A *property-declaration* may include a set of *attributes* (§17), a valid combination of the four access modifiers (§10.3.5), and the `new` (§10.3.4), `static` (§10.6.2), `virtual` (§10.6.3), `override` (§10.6.4), `sealed` (§10.6.5), `abstract` (§10.6.6), and `extern` (§10.6.7) modifiers.

www.it-ebooks.info

Property declarations are subject to the same rules as method declarations (§10.6) with regard to valid combinations of modifiers.

The *type* of a property declaration specifies the type of the property introduced by the declaration, and the *member-name* specifies the name of the property. Unless the property is an explicit interface member implementation, the *member-name* is simply an *identifier*. For an explicit interface member implementation (§13.4.1), the *member-name* consists of an *interface-type* followed by a "`.`" and an *identifier*.

The *type* of a property must be at least as accessible as the property itself (§3.5.4).

The *accessor-declarations*, which must be enclosed in "`{`" and "`}`" tokens, declare the accessors (§10.7.2) of the property. The accessors specify the executable statements associated with reading and writing the property.

Even though the syntax for accessing a property is the same as that for a field, a property is not classified as a variable. Thus it is not possible to pass a property as a `ref` or `out` argument.

When a property declaration includes an `extern` modifier, the property is said to be an ***external property***. Because an external property declaration provides no actual implementation, each of its *accessor-declarations* consists of a semicolon.

10.7.1 Static and Instance Properties

When a property declaration includes a `static` modifier, the property is said to be a ***static property***. When no `static` modifier is present, the property is said to be an ***instance property***.

A static property is not associated with a specific instance, and it is a compile-time error to refer to `this` in the accessors of a static property.

An instance property is associated with a given instance of a class, and that instance can be accessed as `this` (§7.6.7) in the accessors of that property.

When a property is referenced in a *member-access* (§7.6.4) of the form `E.M`, if `M` is a static property, `E` must denote a type containing `M`; if `M` is an instance property, E must denote an instance of a type containing `M`.

The differences between static and instance members are discussed further in §10.3.7.

www.it-ebooks.info

10.7.2 Accessors

The *accessor-declarations* of a property specify the executable statements associated with reading and writing that property.

accessor-declarations:
    get-accessor-declaration *set-accessor-declaration*$_{opt}$
    set-accessor-declaration *get-accessor-declaration*$_{opt}$

get-accessor-declaration:
    attributes$_{opt}$ *accessor-modifier*$_{opt}$ `get` *accessor-body*

set-accessor-declaration:
    attributes$_{opt}$ *accessor-modifier*$_{opt}$ `set` *accessor-body*

accessor-modifier:
    `protected`
    `internal`
    `private`
    `protected internal`
    `internal protected`

accessor-body:
    block
    `;`

The accessor declarations consist of a *get-accessor-declaration*, a *set-accessor-declaration*, or both. Each accessor declaration consists of the token `get` or `set` followed by an optional *accessor-modifier* and an *accessor-body*.

The use of *accessor-modifier*s is governed by the following restrictions:

- An *accessor-modifier* may not be used in an interface or in an explicit interface member implementation.
- For a property or indexer that has no `override` modifier, an *accessor-modifier* is permitted only if the property or indexer has both a `get` and `set` accessor, and then is permitted on only one of those accessors.
- For a property or indexer that includes an `override` modifier, an accessor must match the *accessor-modifier*, if any, of the accessor being overridden.

www.it-ebooks.info

- The *accessor-modifier* must declare an accessibility that is strictly more restrictive than the declared accessibility of the property or indexer itself. To be precise:
  - If the property or indexer has a declared accessibility of `public`, the *accessor-modifier* may be either `protected internal`, `internal`, `protected`, or `private`.
  - If the property or indexer has a declared accessibility of `protected internal`, the *accessor-modifier* may be either `internal`, `protected`, or `private`.
  - If the property or indexer has a declared accessibility of `internal` or `protected`, the *accessor-modifier* must be `private`.
  - If the property or indexer has a declared accessibility of `private`, no *accessor-modifier* may be used.

For `abstract` and `extern` properties, the *accessor-body* for each accessor specified is simply a semicolon. A non-`abstract`, non-`extern` property may be an ***automatically implemented property***, in which case both `get` and `set` accessors must be given, both with a semicolon body (§10.7.3). For the accessors of any other non-`abstract`, non-`extern` property, the *accessor-body* is a *block* that specifies the statements to be executed when the corresponding accessor is invoked.

A `get` accessor corresponds to a parameterless method with a return value of the property type. Except as the target of an assignment, when a property is referenced in an expression, the `get` accessor of the property is invoked to compute the value of the property (§7.1.1). The body of a `get` accessor must conform to the rules for value-returning methods described in §10.6.10. In particular, all `return` statements in the body of a `get` accessor must specify an expression that is implicitly convertible to the property type. Furthermore, the end point of a `get` accessor must not be reachable.

A `set` accessor corresponds to a method with a single value parameter of the property type and a `void` return type. The implicit parameter of a `set` accessor is always named `value`. When a property is referenced as the target of an assignment (§7.17) or as the operand of `++` or `--` (§7.6.9, §7.7.5), the `set` accessor is invoked with an argument (whose value is that of the right-hand side of the assignment or the operand of the `++` or `--` operator) that provides the new value (§7.17.1). The body of a `set` accessor must conform to the rules for `void` methods described in §10.6.10. In particular, `return` statements in the `set` accessor body are not permitted to specify an expression. Since a `set` accessor implicitly has a parameter named `value`, it is a compile-time error for a local variable or constant declaration in a `set` accessor to have that name.

Based on the presence or absence of the `get` and `set` accessors, a property is classified as follows:

www.it-ebooks.info

- A property that includes both a `get` accessor and a `set` accessor is said to be a ***read-write*** property.
- A property that has only a `get` accessor is said to be a ***read-only*** property. It is a compile-time error for a read-only property to be the target of an assignment.
- A property that has only a `set` accessor is said to be a ***write-only*** property. Except as the target of an assignment, it is a compile-time error to reference a write-only property in an expression.

In the example

```
public class Button: Control
{
    private string caption;

    public string Caption {
        get {
            return caption;
        }
        set {
            if (caption != value) {
                caption = value;
                Repaint();
            }
        }
    }

    public override void Paint(Graphics g, Rectangle r) {
        // Painting code goes here
    }
}
```

the `Button` control declares a public `Caption` property. The `get` accessor of the `Caption` property returns the string stored in the private `caption` field. The `set` accessor checks whether the new value is different from the current value, and if so, it stores the new value and repaints the control. Properties often follow the pattern shown above: The `get` accessor simply returns a value stored in a `private` field, and the `set` accessor modifies that `private` field and then performs any additional actions required to fully update the state of the object.

Given the `Button` class above, the following is an example of use of the `Caption` property:

```
Button okButton = new Button();
okButton.Caption = "OK";          // Invokes set accessor
string s = okButton.Caption;      // Invokes get accessor
```

Here, the `set` accessor is invoked by assigning a value to the property, and the `get` accessor is invoked by referencing the property in an expression.

www.it-ebooks.info

The `get` and `set` accessors of a property are not distinct members, and it is not possible to declare the accessors of a property separately. As such, it is not possible for the two accessors of a read-write property to have different accessibility. The example

```
class A
{
    private string name;

    public string Name {        // Error: duplicate member name
        get { return name; }
    }

    public string Name {        // Error: duplicate member name
        set { name = value; }
    }
}
```

does not declare a single read-write property. Rather, it declares two properties with the same name, one read-only and one write-only. Since two members declared in the same class cannot have the same name, the example causes a compile-time error to occur.

When a derived class declares a property by the same name as an inherited property, the derived property hides the inherited property with respect to both reading and writing. In the example

```
class A
{
    public int P {
        set {...}
    }
}

class B: A
{
    new public int P {
        get {...}
    }
}
```

the `P` property in `B` hides the `P` property in `A` with respect to both reading and writing. Thus, in the statements

```
B b = new B();
b.P = 1;              // Error: B.P is read-only
((A)b).P = 1;         // Okay: reference to A.P
```

the assignment to `b.P` causes a compile-time error to be reported, since the read-only `P` property in `B` hides the write-only `P` property in `A`. Note, however, that a cast can be used to access the hidden `P` property.

www.it-ebooks.info

Unlike `public` fields, properties provide a separation between an object's internal state and its `public` interface. Consider the following example:

```
class Label
{
    private int x, y;
    private string caption;

    public Label(int x, int y, string caption) {
        this.x = x;
        this.y = y;
        this.caption = caption;
    }

    public int X {
        get { return x; }
    }

    public int Y {
        get { return y; }
    }

    public Point Location {
        get { return new Point(x, y); }
    }

    public string Caption {
        get { return caption; }
    }
}
```

Here, the `Label` class uses two `int` fields, `x` and `y`, to store its location. The location is publicly exposed both as an `X` and a `Y` property and as a `Location` property of type `Point`. If, in a future version of `Label`, it becomes more convenient to store the location as a `Point` internally, the change can be made without affecting the `public` interface of the class:

```
class Label
{
    private Point location;
    private string caption;

    public Label(int x, int y, string caption) {
        this.location = new Point(x, y);
        this.caption = caption;
    }

    public int X {
        get { return location.x; }
    }

    public int Y {
        get { return location.y; }
    }
```


www.it-ebooks.info

```
    public Point Location {
        get { return location; }
    }

    public string Caption {
        get { return caption; }
    }
}
```

Had `x` and `y` instead been `public readonly` fields, it would have been impossible to make such a change to the `Label` class.

Exposing state through properties is not necessarily any less efficient than exposing fields directly. In particular, when a property is non-virtual and contains only a small amount of code, the execution environment may replace calls to accessors with the actual code of the accessors. This process is known as ***inlining***, and it makes property access as efficient as field access, yet preserves the increased flexibility of properties.

Since invoking a `get` accessor is conceptually equivalent to reading the value of a field, it is considered bad programming style for `get` accessors to have observable side effects. In the example

```
class Counter
{
    private int next;

    public int Next {
        get { return next++; }
    }
}
```

the value of the `Next` property depends on the number of times the property has previously been accessed. Thus accessing the property produces an observable side effect, and the property should be implemented as a method instead.

The "no side effects" convention for `get` accessors doesn't mean that `get` accessors should always be written to simply return values stored in fields. Indeed, `get` accessors often compute the value of a property by accessing multiple fields or invoking methods. However, a properly designed `get` accessor performs no actions that cause observable changes in the state of the object.

Properties can be used to delay initialization of a resource until the moment it is first referenced. For example:

```
using System.IO;

public class Console
{
    private static TextReader reader;
    private static TextWriter writer;
    private static TextWriter error;
```


www.it-ebooks.info

```
        public static TextReader In {
            get {
                if (reader == null) {
                    reader = new StreamReader(Console.OpenStandardInput());
                }
                return reader;
            }
        }

        public static TextWriter Out {
            get {
                if (writer == null) {
                    writer = new StreamWriter(Console.OpenStandardOutput());
                }
                return writer;
            }
        }

        public static TextWriter Error {
            get {
                if (error == null) {
                    error = new StreamWriter(Console.OpenStandardError());
                }
                return error;
            }
        }
    }
```

The `Console` class contains three properties—`In`, `Out`, and `Error`—that represent the standard input, output, and error devices, respectively. By exposing these members as properties, the `Console` class can delay their initialization until they are actually used. For example, upon first referencing the `Out` property, as in

```
Console.Out.WriteLine("hello, world");
```

the underlying `TextWriter` for the output device is created. But if the application makes no reference to the `In` and `Error` properties, then no objects are created for those devices.

10.7.3 Automatically Implemented Properties

> **JON SKEET** An early draft of the C# 3.0 specification referred to these properties as "automatic properties." This name was around for just long enough to stick, and is now much more widely used in the community than the full name.

When a property is specified as an automatically implemented property, a hidden backing field is automatically available for the property, and the accessors are implemented to read from and write to that backing field.

www.it-ebooks.info

The example

```
public class Point {
    public int X { get; set; } // Automatically implemented
    public int Y { get; set; } // Automatically implemented
}
```

is equivalent to the following declaration:

```
public class Point {
    private int x;
    private int y;
    public int X { get { return x; } set { x = value; } }
    public int Y { get { return y; } set { y = value; } }
}
```


Because the backing field is inaccessible, it can be read and written only through the property accessors, even within the containing type. This means that automatically implemented read-only or write-only properties do not make sense and, therefore, are disallowed. It is possible to set the access level of each accessor differently. Thus the effect of a read-only property with a private backing field can be mimicked like this:

```
public class ReadOnlyPoint {
    public int X { get; private set; }
    public int Y { get; private set; }
    public ReadOnlyPoint(int x, int y) { X = x; Y = y; }
}
```

BILL WAGNER While the properties are read-only, the underlying backing store is not. Code inside `ReadOnlyPoint` does not have to obey the rules of immutability.

JON SKEET While this behavior mimics the effect of a read-only property to other types, it's not the same as the property (or, indeed, the backing field) being genuinely read-only. I would welcome the ability to specify read-only properties, perhaps in this form:

```
public int X { get; readonly set; }
```

This code would result in a read-only field being produced; use of the setter would be permitted only within the constructor, where it would be compiled into direct writes to the backing field. It's unfortunate that creating a genuinely immutable type is currently more long-winded than creating a mutable one.

www.it-ebooks.info

PETER SESTOFT Automatically implemented properties are great because the properties can be used in object initializers (§7.6.10.2). Object initializers are particularly great for classical values such as coordinate pairs and triples, complex numbers, and similar items—that is, struct values. In general, however, we prefer structs to have immutable fields because they are copied on assignment and during parameter passing. For this reason, I whole-heartedly support Jon's wish for a terse notation for read-only fields with automatically implemented accessors, with the proviso that the `set` accessor can be used in the constructor *and in object initializers.*

This restriction also means that definite assignment of struct types with auto-implemented properties can be achieved only using the standard constructor of the struct, since assigning to the property itself requires the struct to be definitely assigned. This means that user-defined constructors must call the default constructor.

10.7.4 Accessibility

If an accessor has an *accessor-modifier*, the accessibility domain (§3.5.2) of the accessor is determined using the declared accessibility of the *accessor-modifier*. If an accessor does not have an *accessor-modifier*, the accessibility domain of the accessor is determined from the declared accessibility of the property or indexer.

The presence of an *accessor-modifier* never affects member lookup (§7.3) or overload resolution (§7.5.3). The modifiers on the property or indexer always determine which property or indexer is bound to, regardless of the context of the access.

Once a particular property or indexer has been selected, the accessibility domains of the specific accessors involved are used to determine whether that usage is valid:

- If the usage is as a value (§7.1.1), the `get` accessor must exist and be accessible.
- If the usage is as the target of a simple assignment (§7.17.1), the `set` accessor must exist and be accessible.
- If the usage is as the target of compound assignment (§7.17.2) or as the target of the `++` or `--` operators (§7.5.9, §7.6.5), both the `get` accessor and the `set` accessor must exist and be accessible.

In the following example, the property `A.Text` is hidden by the property `B.Text`, even in contexts where only the `set` accessor is called. In contrast, the property `B.Count` is not accessible to class `M`, so the accessible property `A.Count` is used instead.

www.it-ebooks.info

```
class A
{
    public string Text {
        get { return "hello"; }
        set { }
    }

    public int Count {
        get { return 5; }
        set { }
    }
}

class B: A
{
    private string text = "goodbye";
    private int count = 0;

    new public string Text {
        get { return text; }
        protected set { text = value; }
    }

    new protected int Count {
        get { return count; }
        set { count = value; }
    }
}

class M
{
    static void Main() {
        B b = new B();
        b.Count = 12;           // Calls A.Count set accessor
        int i = b.Count;        // Calls A.Count get accessor
        b.Text = "howdy";       // Error: B.Text set accessor is not accessible
        string s = b.Text;      // Calls B.Text get accessor
    }
}
```

An accessor that is used to implement an interface may not have an *accessor-modifier*. If only one accessor is used to implement an interface, the other accessor may be declared with an *accessor-modifier*:

```
public interface I
{
    string Prop { get; }
}

public class C: I
{
    public Prop {
        get { return "April"; } // Must not have a modifier here
        internal set {...}      // Okay, because I.Prop has no set accessor
    }
}
```


www.it-ebooks.info

10.7.5 Virtual, Sealed, Override, and Abstract Accessors

A `virtual` property declaration specifies that the accessors of the property are virtual. The `virtual` modifier applies to both accessors of a read-write property—it is not possible for only one accessor of a read-write property to be virtual.

An `abstract` property declaration specifies that the accessors of the property are virtual, but does not provide an actual implementation of the accessors. Instead, non-`abstract` derived classes are required to provide their own implementation for the accessors by overriding the property. Because an accessor for an `abstract` property declaration provides no actual implementation, its *accessor-body* simply consists of a semicolon.

A property declaration that includes both the `abstract` and `override` modifiers specifies that the property is abstract and overrides a base property. The accessors of such a property are also abstract.

Abstract property declarations are permitted only in abstract classes (§10.1.1.1). The accessors of an inherited virtual property can be overridden in a derived class by including a property declaration that specifies an `override` directive. This is known as an ***overriding property declaration***. An overriding property declaration does not declare a new property. Instead, it simply specializes the implementations of the accessors of an existing virtual property.

An overriding property declaration must specify the exact same accessibility modifiers, type, and name as the inherited property. If the inherited property has only a single accessor (i.e., if the inherited property is read-only or write-only), the overriding property must include only that accessor. If the inherited property includes both accessors (i.e., if the inherited property is read-write), the overriding property can include either a single accessor or both accessors.

An overriding property declaration may include the `sealed` modifier. Use of this modifier prevents a derived class from further overriding the property. The accessors of a sealed property are also sealed.

Except for differences in declaration and invocation syntax, virtual, sealed, override, and abstract accessors behave exactly like virtual, sealed, override, and abstract methods. Specifically, the rules described in §10.6.3, §10.6.4, §10.6.5, and §10.6.6 apply as if accessors were methods of a corresponding form:

- A `get` accessor corresponds to a parameterless method with a return value of the property type and the same modifiers as the containing property.
- A `set` accessor corresponds to a method with a single value parameter of the property type, a `void` return type, and the same modifiers as the containing property.

www.it-ebooks.info

In the example

```
abstract class A
{
    int y;

    public virtual int X {
        get { return 0; }
    }

    public virtual int Y {
        get { return y; }
        set { y = value; }
    }

    public abstract int Z { get; set; }
}
```

`X` is a virtual read-only property, `Y` is a virtual read-write property, and `Z` is an `abstract` read-write property. Because `Z` is `abstract`, the containing class `A` must also be declared as `abstract`.

A class that derives from `A` is show below:

```
class B: A
{
    int z;

    public override int X {
        get { return base.X + 1; }
    }

    public override int Y {
        set { base.Y = value < 0? 0: value; }
    }

    public override int Z {
        get { return z; }
        set { z = value; }
    }
}
```

Here, the declarations of `X`, `Y`, and `Z` are overriding property declarations. Each property declaration exactly matches the accessibility modifiers, type, and name of the corresponding inherited property. The `get` accessor of `X` and the `set` accessor of `Y` use the `base` keyword to access the inherited accessors. The declaration of `Z` overrides both abstract accessors—thus there are no outstanding abstract function members in `B`, and `B` is permitted to be a nonabstract class.

When a property is declared as an `override`, any overridden accessors must be accessible to the overriding code. In addition, the declared accessibility of both the property or indexer itself, and of the accessors, must match that of the overridden member and accessors. For example:

www.it-ebooks.info

```
public class B
{
    public virtual int P {
        protected set {...}
        get {...}
    }
}

public class D: B
{
    public override int P {
        protected set {...}     // Must specify protected here
        get {...}               // Must not have a modifier here
    }
}
```

10.8 Events

An ***event*** is a member that enables an object or class to provide notifications. Clients can attach executable code for events by supplying ***event handlers***.

Events are declared using *event-declaration*s:

event-declaration:
    attributes$_{opt}$ *event-modifiers*$_{opt}$ `event` *type* *variable-declarators* `;`
    attributes$_{opt}$ *event-modifiers*$_{opt}$ `event` *type* *member-name*
        `{` *event-accessor-declarations* `}`

event-modifiers:
    event-modifier
    event-modifiers *event-modifier*

event-modifier:
    `new`
    `public`
    `protected`
    `internal`
    `private`
    `static`
    `virtual`
    `sealed`
    `override`
    `abstract`
    `extern`

www.it-ebooks.info

event-accessor-declarations:
    add-accessor-declaration *remove-accessor-declaration*
    remove-accessor-declaration *add-accessor-declaration*

add-accessor-declaration:
    attributes$_{opt}$ `add` *block*

remove-accessor-declaration:
    attributes$_{opt}$ `remove` *block*

An *event-declaration* may include a set of *attributes* (§17), a valid combination of the four access modifiers (§10.3.5), and the `new` (§10.3.4), `static` (§10.6.2), `virtual` (§10.6.3), `override` (§10.6.4), `sealed` (§10.6.5), `abstract` (§10.6.6), and `extern` (§10.6.7) modifiers.

Event declarations are subject to the same rules as method declarations (§10.6) with regard to valid combinations of modifiers.

The *type* of an event declaration must be a *delegate-type* (§4.2), and that *delegate-type* must be at least as accessible as the event itself (§3.5.4).

An event declaration may include *event-accessor-declaration*s. However, if it does not, for non-`extern`, non-`abstract` events, the compiler supplies them automatically (§10.8.1); for `extern` events, the accessors are provided externally.

An event declaration that omits *event-accessor-declaration*s defines one or more events—one for each of the *variable-declarator*s. The attributes and modifiers apply to all of the members declared by such an *event-declaration*.

It is a compile-time error for an *event-declaration* to include both the `abstract` modifier and brace-delimited *event-accessor-declaration*s.

When an event declaration includes an `extern` modifier, the event is said to be an ***external event***. Because an external event declaration provides no actual implementation, it is an error for it to include both the `extern` modifier and *event-accessor-declaration*s.

An event can be used as the left-hand operand of the `+=` and `-=` operators (§7.17.3). These operators are used, respectively, to attach event handlers to or to remove event handlers from an event, and the access modifiers of the event control the contexts in which such operations are permitted.

Since `+=` and `-=` are the only operations that are permitted on an event outside the type that declares the event, external code can add and remove handlers for an event, but cannot in any other way obtain or modify the underlying list of event handlers.

www.it-ebooks.info

In an operation of the form `x += y` or `x -= y`, when `x` is an event and the reference takes place outside the type that contains the declaration of `x`, the result of the operation has type `void` (as opposed to having the type of `x`, with the value of `x` after the assignment). This rule prohibits external code from indirectly examining the underlying delegate of an event.

The following example shows how event handlers are attached to instances of the `Button` class:

```
public delegate void EventHandler(object sender, EventArgs e);

public class Button: Control
{
    public event EventHandler Click;
}

public class LoginDialog: Form
{
    Button OkButton;
    Button CancelButton;

    public LoginDialog() {
        OkButton = new Button(...);
        OkButton.Click += new EventHandler(OkButtonClick);
        CancelButton = new Button(...);
        CancelButton.Click += new EventHandler(CancelButtonClick);
    }

    void OkButtonClick(object sender, EventArgs e) {
        // Handle OkButton.Click event
    }

    void CancelButtonClick(object sender, EventArgs e) {
        // Handle CancelButton.Click event
    }
}
```

Here, the `LoginDialog` instance constructor creates two `Button` instances and attaches event handlers to the `Click` events.

CHRISTIAN NAGEL Memory leaks often result from wrong usage of events. If client objects attach to events but do not detach from them, and the reference to the client object is no longer used, the client object still cannot be reclaimed by the garbage collector because the reference by the publisher remains. This can be avoided by (1) detaching of events when the client object is no longer used, (2) a custom implementation of the add and remove accessors using the `WeakReference` class holding the delegate, or (3) the Weak Event pattern that is used by WPF with the `IWeakEventListener` interface.

www.it-ebooks.info

> **JON SKEET** I tend to think of (and explain) events as being like properties: They're a pair of methods (add and remove) that the compiler knows how to call using a shorthand (`+=` and `-=`). Events are more restrictive than properties in that the type has to be a delegate type, and there must *always* be both add and remove methods; there is no such thing as an "add-only" event in the way that you can have a read-only property. Other than those points, the similarities are strong—and yet properties are generally very well understood, whereas events cause heaps of confusion. I surmise that field-like events are the source of this problem.

10.8.1 Field-like Events

Within the program text of the class or struct that contains the declaration of an event, certain events can be used like fields. To be used in this way, an event must not be `abstract` or `extern`, and must not explicitly include *event-accessor-declaration*s. Such an event can be used in any context that permits a field. The field contains a delegate (§15), which refers to the list of event handlers that have been added to the event. If no event handlers have been added, the field contains `null`.

> **JON SKEET** If automatically implemented properties had been present in C# from the start, I believe it would have made more sense for field-like events to be called "automatically implemented events" with a syntax like this:
>
> ```
> public event EventHandler Click { add; remove; }
> ```
>
> This might have created less confusion around what an event really is.

In the example

```
public delegate void EventHandler(object sender, EventArgs e);

public class Button: Control
{
    public event EventHandler Click;

    protected void OnClick(EventArgs e) {
        if (Click != null) Click(this, e);
    }

    public void Reset() {
        Click = null;
    }
}
```


www.it-ebooks.info

`Click` is used as a field within the `Button` class. As the example demonstrates, the field can be examined, modified, and used in delegate invocation expressions. The `OnClick` method in the `Button` class "raises" the `Click` event. The notion of raising an event is precisely equivalent to invoking the delegate represented by the event—thus there are no special language constructs for raising events. Note that the delegate invocation is preceded by a check that ensures the delegate is non-null.

Outside the declaration of the `Button` class, the `Click` member can be used only on the left-hand side of the `+=` and `-=` operators, as in

```
b.Click += new EventHandler(...);
```

which appends a delegate to the invocation list of the `Click` event, and

```
b.Click -= new EventHandler(...);
```

which removes a delegate from the invocation list of the `Click` event.

When compiling a field-like event, the compiler automatically creates storage to hold the delegate, and creates accessors for the event that add or remove event handlers to the delegate field. The addition and removal operations are thread safe, and may (but are not required to) be done while holding the lock (§8.12) on the containing object for an instance event, or the type object (§7.6.10.6) for a static event.

Thus an instance event declaration of the form

```
class X
{
    public event D Ev;
}
```

will be compiled to something equivalent to

```
class X
{
    private D __Ev;  // Field to hold the delegate

    public event D Ev {
        add {
            /* Add the delegate in a thread-safe way */
        }

        remove {
            /* Remove the delegate in a thread-safe way */
        }
    }
}
```


www.it-ebooks.info

Within the class `X`, references to `Ev` on the left-hand side of the `+=` and `-=` operators cause the add and remove accessors to be invoked. All other references to `Ev` are compiled to reference the hidden field `__Ev` instead. The name "`__Ev`" is arbitrary; the hidden field could have any name or no name at all.

10.8.2 Event Accessors

Event declarations typically omit *event-accessor-declaration*s, as in the `Button` example above. One situation for doing so involves the case in which the storage cost of one field per event is not acceptable. In such cases, a class can include *event-accessor-declaration*s and use a private mechanism for storing the list of event handlers.

The *event-accessor-declarations* of an event specify the executable statements associated with adding and removing event handlers.

The accessor declarations consist of an *add-accessor-declaration* and a *remove-accessor-declaration*. Each accessor declaration consists of the token `add` or `remove` followed by a *block*. The *block* associated with an *add-accessor-declaration* specifies the statements to execute when an event handler is added, and the *block* associated with a *remove-accessor-declaration* specifies the statements to execute when an event handler is removed.

Each *add-accessor-declaration* and *remove-accessor-declaration* corresponds to a method with a single value parameter of the event type and a `void` return type. The implicit parameter of an event accessor is named `value`. When an event is used in an event assignment, the appropriate event accessor is used. Specifically, if the assignment operator is `+=`, then the add accessor is used; if the assignment operator is `-=`, then the remove accessor is used. In either case, the right-hand operand of the assignment operator is used as the argument to the event accessor. The block of an *add-accessor-declaration* or a *remove-accessor-declaration* must conform to the rules for `void` methods described in §10.6.10. In particular, `return` statements in such a block are not permitted to specify an expression.

Since an event accessor implicitly has a parameter named `value`, it is a compile-time error for a local variable or constant declared in an event accessor to have that name.

In the example

```
class Control: Component
{
    // Unique keys for events
    static readonly object mouseDownEventKey = new object();
    static readonly object mouseUpEventKey = new object();

    // Return event handler associated with key
    protected Delegate GetEventHandler(object key) {...}

    // Add event handler associated with key
    protected void AddEventHandler(object key, Delegate handler) {...}
```


www.it-ebooks.info

```
        // Remove event handler associated with key
        protected void RemoveEventHandler(object key, Delegate handler) {...}

        // MouseDown event
        public event MouseEventHandler MouseDown {
            add { AddEventHandler(mouseDownEventKey, value); }
            remove { RemoveEventHandler(mouseDownEventKey, value); }
        }

        // MouseUp event
        public event MouseEventHandler MouseUp {
            add { AddEventHandler(mouseUpEventKey, value); }
            remove { RemoveEventHandler(mouseUpEventKey, value); }
        }

        // Invoke the MouseUp event
        protected void OnMouseUp(MouseEventArgs args) {
            MouseEventHandler handler;
            handler = (MouseEventHandler)GetEventHandler(mouseUpEventKey);
            if (handler != null)
                handler(this, args);
        }
    }
```

the `Control` class implements an internal storage mechanism for events. The `AddEventHandler` method associates a delegate value with a key, the `GetEventHandler` method returns the delegate currently associated with a key, and the `RemoveEventHandler` method removes a delegate as an event handler for the specified event. Presumably, the underlying storage mechanism is designed such that there is no cost for associating a `null` delegate value with a key, and thus unhandled events consume no storage.

10.8.3 Static and Instance Events

When an event declaration includes a `static` modifier, the event is said to be a ***static event***. When no `static` modifier is present, the event is said to be an ***instance event***.

A static event is not associated with a specific instance, and it is a compile-time error to refer to `this` in the accessors of a static event.

An instance event is associated with a given instance of a class, and this instance can be accessed as `this` (§7.6.7) in the accessors of that event.

When an event is referenced in a *member-access* (§7.6.4) of the form `E.M`, if `M` is a static event, `E` must denote a type containing `M`; if `M` is an instance event, E must denote an instance of a type containing `M`.

The differences between static and instance members are discussed further in §10.3.7.

www.it-ebooks.info

10.8.4 Virtual, Sealed, Override, and Abstract Accessors

A `virtual` event declaration specifies that the accessors of that event are virtual. The `virtual` modifier applies to both accessors of an event.

An `abstract` event declaration specifies that the accessors of the event are virtual, but does not provide an actual implementation of the accessors. Instead, non-`abstract` derived classes are required to provide their own implementation for the accessors by overriding the event. Because an `abstract` event declaration provides no actual implementation, it cannot provide brace-delimited *event-accessor-declaration*s.

An event declaration that includes both the `abstract` and `override` modifiers specifies that the event is abstract and overrides a base event. The accessors of such an event are also abstract.

Abstract event declarations are permitted only in abstract classes (§10.1.1.1).

The accessors of an inherited virtual event can be overridden in a derived class by including an event declaration that specifies an `override` modifier. This is known as an ***overriding event declaration***. An overriding event declaration does not declare a new event. Instead, it simply specializes the implementations of the accessors of an existing virtual event.

An overriding event declaration must specify the exact same accessibility modifiers, type, and name as the overridden event.

An overriding event declaration may include the `sealed` modifier. Use of this modifier prevents a derived class from further overriding the event. The accessors of a sealed event are also sealed.

It is a compile-time error for an overriding event declaration to include a `new` modifier.

Except for differences in declaration and invocation syntax, virtual, sealed, override, and abstract accessors behave exactly like virtual, sealed, override, and abstract methods. Specifically, the rules described in §10.6.3, §10.6.4, §10.6.5, and §10.6.6 apply as if accessors were methods of a corresponding form. Each accessor corresponds to a method with a single value parameter of the event type, a `void` return type, and the same modifiers as the containing event.

10.9 Indexers

An ***indexer*** is a member that enables an object to be indexed in the same way as an array. Indexers are declared using *indexer-declaration*s:

www.it-ebooks.info

indexer-declaration:
    attributes$_{opt}$ *indexer-modifiers*$_{opt}$ *indexer-declarator* `{` *accessor-declarations* `}`

indexer-modifiers:
    indexer-modifier
    indexer-modifiers *indexer-modifier*

indexer-modifier:
    `new`
    `public`
    `protected`
    `internal`
    `private`
    `virtual`
    `sealed`
    `override`
    `abstract`
    `extern`

indexer-declarator:
    type `this` `[` *formal-parameter-list* `]`
    type *interface-type* `.` `this` `[` *formal-parameter-list* `]`

An *indexer-declaration* may include a set of *attributes* (§17), a valid combination of the four access modifiers (§10.3.5), and the `new` (§10.3.4), `virtual` (§10.6.3), `override` (§10.6.4), `sealed` (§10.6.5), `abstract` (§10.6.6), and `extern` (§10.6.7) modifiers.

Indexer declarations are subject to the same rules as method declarations (§10.6) with regard to valid combinations of modifiers, with the one exception being that the `static` modifier is not permitted on an indexer declaration.

The modifiers `virtual`, `override`, and `abstract` are mutually exclusive except in one case. The `abstract` and `override` modifiers may be used together so that an abstract indexer can override a virtual one.

The *type* of an indexer declaration specifies the element type of the indexer introduced by the declaration. Unless the indexer is an explicit interface member implementation, the *type* is followed by the keyword `this`. For an explicit interface member implementation, the *type* is followed by an *interface-type*, a "`.`", and the keyword `this`. Unlike other members, indexers do not have user-defined names.

The *formal-parameter-list* specifies the parameters of the indexer. The formal parameter list of an indexer corresponds to that of a method (§10.6.1), except that at least one parameter must be specified, and that the `ref` and `out` parameter modifiers are not permitted.

www.it-ebooks.info

The *type* of an indexer and each of the types referenced in the *formal-parameter-list* must be at least as accessible as the indexer itself (§3.5.4).

The *accessor-declarations* (§10.7.2), which must be enclosed in "`{`" and "`}`" tokens, declare the accessors of the indexer. The accessors specify the executable statements associated with reading and writing indexer elements.

Even though the syntax for accessing an indexer element is the same as that for an array element, an indexer element is not classified as a variable. Thus it is not possible to pass an indexer element as a `ref` or `out` argument.

The formal parameter list of an indexer defines the signature (§3.6) of the indexer. Specifically, the signature of an indexer consists of the number and types of its formal parameters. The element type and names of the formal parameters are not part of an indexer's signature.

The signature of an indexer must differ from the signatures of all other indexers declared in the same class.

Indexers and properties are very similar in concept, but differ in the following ways:

- A property is identified by its name, whereas an indexer is identified by its signature.
- A property is accessed through a *simple-name* (§7.6.2) or a *member-access* (§7.6.4), whereas an indexer element is accessed through an *element-access* (§7.6.6.2).
- A property can be a `static` member, whereas an indexer is always an instance member.
- A `get` accessor of a property corresponds to a method with no parameters, whereas a `get` accessor of an indexer corresponds to a method with the same formal parameter list as the indexer.
- A `set` accessor of a property corresponds to a method with a single parameter named `value`, whereas a `set` accessor of an indexer corresponds to a method with the same formal parameter list as the indexer, plus an additional parameter named `value`.
- It is a compile-time error for an indexer accessor to declare a local variable with the same name as an indexer parameter.
- In an overriding property declaration, the inherited property is accessed using the syntax `base.P`, where `P` is the property name. In an overriding indexer declaration, the inherited indexer is accessed using the syntax `base[E]`, where `E` is a comma-separated list of expressions.

Aside from these differences, all rules defined in §10.7.2 and §10.7.3 apply to indexer accessors as well as to property accessors.

www.it-ebooks.info

JON SKEET I find it odd that events can be static (but almost never are) but indexers can't. For example, it might have made sense for `Encoding.GetEncoding(string)` to be present as a static indexer. Indeed, I'd expect static indexers to primarily act as factory members for the type they're declared in. A method is a perfectly adequate alternative in most places, but the prohibition is a little strange.

When an indexer declaration includes an `extern` modifier, the indexer is said to be an ***external indexer***. Because an external indexer declaration provides no actual implementation, each of its *accessor-declarations* consists of a semicolon.

The example below declares a `BitArray` class that implements an indexer for accessing the individual bits in the bit array.

```
using System;

class BitArray
{
    int[] bits;
    int length;

    public BitArray(int length) {
        if (length < 0) throw new ArgumentException();
        bits = new int[((length - 1) >> 5) + 1];
        this.length = length;
    }

    public int Length {
        get { return length; }
    }

    public bool this[int index] {
        get {
            if (index < 0 || index >= length) {
                throw new IndexOutOfRangeException();
            }
            return (bits[index >> 5] & 1 << index) != 0;
        }
        set {
            if (index < 0 || index >= length) {
                throw new IndexOutOfRangeException();
            }
            if (value) {
                bits[index >> 5] |= 1 << index;
            }
            else {
                bits[index >> 5] &= ~(1 << index);
            }
        }
    }
}
```


www.it-ebooks.info

An instance of the `BitArray` class consumes substantially less memory than a corresponding `bool[]` (since each value of the former occupies only one bit instead of the latter's one byte), but it permits the same operations as a `bool[]`.

The following `CountPrimes` class uses a `BitArray` and the classical "sieve" algorithm to compute the number of primes between 1 and a given maximum:

```
class CountPrimes
{
    static int Count(int max) {
        BitArray flags = new BitArray(max + 1);
        int count = 1;
        for (int i = 2; i <= max; i++) {
            if (!flags[i]) {
                for (int j = i * 2; j <= max; j += i) flags[j] = true;
                count++;
            }
        }
        return count;
    }

    static void Main(string[] args) {
        int max = int.Parse(args[0]);
        int count = Count(max);
        Console.WriteLine("Found {0} primes between 1 and {1}", count, max);
    }
}
```

Note that the syntax for accessing elements of the `BitArray` is precisely the same as for a `bool[]`.

The following example shows a 26 × 10 grid class that has an indexer with two parameters. The first parameter is required to be an uppercase or lowercase letter in the range A–Z, and the second is required to be an integer in the range 0–9.

```
using System;

class Grid
{
    const int NumRows = 26;
    const int NumCols = 10;

    int[,] cells = new int[NumRows, NumCols];

    public int this[char c, int col] {
        get {
            c = Char.ToUpper(c);
            if (c < 'A' || c > 'Z') {
                throw new ArgumentException();
            }
```


www.it-ebooks.info

```
            if (col < 0 || col >= NumCols) {
                throw new IndexOutOfRangeException();
            }
            return cells[c - 'A', col];
        }

        set {
            c = Char.ToUpper(c);
            if (c < 'A' || c > 'Z') {
                throw new ArgumentException();
            }
            if (col < 0 || col >= NumCols) {
                throw new IndexOutOfRangeException();
            }
            cells[c - 'A', col] = value;
        }
    }
}
```

10.9.1 Indexer Overloading

The indexer overload resolution rules are described in §7.5.2.

10.10 Operators

An ***operator*** is a member that defines the meaning of an expression operator that can be applied to instances of the class. Operators are declared using *operator-declaration*s:

operator-declaration:
    attributes$_{opt}$ *operator-modifiers* *operator-declarator* *operator-body*

operator-modifiers:
    operator-modifier
    operator-modifiers *operator-modifier*

operator-modifier:
    `public`
    `static`
    `extern`

operator-declarator:
    unary-operator-declarator
    binary-operator-declarator
    conversion-operator-declarator

www.it-ebooks.info

unary-operator-declarator:
    type `operator` *overloadable-unary-operator* `(` *type identifier* `)`

overloadable-unary-operator: one of
    `+` `-` `!` `~` `++` `--` `true` `false`

binary-operator-declarator:
    type `operator` *overloadable-binary-operator* `(` *type identifier* `,` *type identifier* `)`

overloadable-binary-operator:
    `+`
    `-`
    `*`
    `/`
    `%`
    `&`
    `|`
    `^`
    `<<`
    right-shift
    `==`
    `!=`
    `>`
    `<`
    `>=`
    `<=`

conversion-operator-declarator:
    `implicit operator` *type* `(` *type identifier* `)`
    `explicit operator` *type* `(` *type identifier* `)`

operator-body:
    block
    `;`

There are three categories of overloadable operators: unary operators (§10.10.1), binary operators (§10.10.2), and conversion operators (§10.10.3).

When an operator declaration includes an `extern` modifier, the operator is said to be an ***external operator***. Because an external operator provides no actual implementation, its *operator-body* consists of a semicolon. For all other operators, the *operator-body* consists of a *block*, which specifies the statements to execute when the operator is invoked. The *block* of an operator must conform to the rules for value-returning methods described in §10.6.10.

www.it-ebooks.info

The following rules apply to all operator declarations:

- An operator declaration must include both a `public` and a `static` modifier.
- The parameter(s) of an operator must be value parameters. It is a compile-time error for an operator declaration to specify `ref` or `out` parameters.
- The signature of an operator (§10.10.1, §10.10.2, §10.10.3) must differ from the signatures of all other operators declared in the same class.
- All types referenced in an operator declaration must be at least as accessible as the operator itself (§3.5.4).
- It is an error for the same modifier to appear multiple times in an operator declaration.

Each operator category imposes additional restrictions, as described in the following sections.

Like other members, operators declared in a base class are inherited by derived classes. Because operator declarations always require the class or struct in which the operator is declared to participate in the signature of the operator, it is not possible for an operator declared in a derived class to hide an operator declared in a base class. Thus the `new` modifier is never required, and therefore never permitted, in an operator declaration.

Additional information on unary and binary operators can be found in §7.3.

Additional information on conversion operators can be found in §6.4.

10.10.1 Unary Operators

The following rules apply to unary operator declarations, where `T` denotes the instance type of the class or struct that contains the operator declaration:

- A unary `+`, `-`, `!`, or `~` operator must take a single parameter of type `T` or `T?` and can return any type.
- A unary `++` or `--` operator must take a single parameter of type `T` or `T?` and must return that same type or a type derived from it.
- A unary `true` or `false` operator must take a single parameter of type `T` or `T?` and must return type `bool`.

The signature of a unary operator consists of the operator token (`+`, `-`, `!`, `~`, `++`, `--`, `true`, or `false`) and the type of the single formal parameter. The return type is not part of a unary operator's signature, nor is the name of the formal parameter.

www.it-ebooks.info

The `true` and `false` unary operators require pairwise declaration. A compile-time error occurs if a class declares one of these operators without also declaring the other. The `true` and `false` operators are described further in §7.12.2 and §7.20.

The following example shows an implementation and subsequent usage of `operator ++` for an integer vector class:

```
public class IntVector
{
    public IntVector(int length) {...}

    public int Length {...}                 // Read-only property

    public int this[int index] {...}        // Read-write indexer

    public static IntVector operator ++(IntVector iv) {
        IntVector temp = new IntVector(iv.Length);
        for (int i = 0; i < iv.Length; i++)
            temp[i] = iv[i] + 1;
        return temp;
    }
}

class Test
{
    static void Main() {
        IntVector iv1 = new IntVector(4);   // Vector of 4 x 0
        IntVector iv2;

        iv2 = iv1++;                        // iv2 contains 4 x 0, iv1 contains 4 x 1
        iv2 = ++iv1;                        // iv2 contains 4 x 2, iv1 contains 4 x 2
    }
}
```

Note how the operator method returns the value produced by adding 1 to the operand, just like the postfix increment and decrement operators (§7.6.9) and the prefix increment and decrement operators (§7.7.5). Unlike in C++, this method need not modify the value of its operand directly. In fact, modifying the operand value would violate the standard semantics of the postfix increment operator.

10.10.2 Binary Operators

The following rules apply to binary operator declarations, where `T` denotes the instance type of the class or struct that contains the operator declaration:

- A binary nonshift operator must take two parameters, at least one of which must have type `T` or `T?`, and can return any type.
- A binary `<<` or `>>` operator must take two parameters, the first of which must have type `T` or `T?` and the second of which must have type `int` or `int?`, and can return any type.

www.it-ebooks.info

The signature of a binary operator consists of the operator token (+, -, *, /, %, &, |, ^, <<, >>, ==, !=, >, <, >=, or <=) and the types of the two formal parameters. The return type and the names of the formal parameters are not part of a binary operator's signature.

Certain binary operators require pairwise declaration. For every declaration of either operator of a pair, there must be a matching declaration of the other operator of the pair. Two operator declarations match when they have the same return type and the same type for each parameter. The following operators require pairwise declaration:

- `operator ==` and `operator !=`
- `operator >` and `operator <`
- `operator >=` and `operator <=`

ERIC LIPPERT It is tempting to make a comparison operator on class `Clothing` that can represent `Socks`, `Shoes`, `Shirt`, `Tie`, and `Hat`, and then define a user-defined comparison that makes `Socks < Shoes, Shirt < Tie`, and everything else equal. The hope is that sorting an array of these items will produce a sensible ordering—that `Socks` always go on before `Shoes`, `Shirts` go on before `Ties`, but everything else can be in any order. But this comparison is not logically consistent: If `Hat` equals `Socks`, `Hat` equals `Shoes`, and `Socks` is smaller than `Shoes` then logically `Hat` must be smaller than `Hat`, which is nonsensical. Most sorting algorithms are written with the assumption that the comparison operator defines a *total ordering*; some algorithms crash, run forever, or give nonsensical results when given a broken comparison operator. If you want to implement a *partial order sort*, it is wise to do it via some mechanism other than overloading the comparison operators.

10.10.3 Conversion Operators

A conversion operator declaration introduces a ***user-defined conversion*** (§6.4), which augments the predefined implicit and explicit conversions.

A conversion operator declaration that includes the `implicit` keyword introduces a user-defined implicit conversion. Implicit conversions can occur in a variety of situations, including function member invocations, cast expressions, and assignments. This is described further in §6.1.

BILL WAGNER Implicit conversions should always succeed and never lose information because they will be called automatically, without the client coders knowing it. Implicit conversions should never involve expensive operations for the same reason.

www.it-ebooks.info

> **JON SKEET** As noted in §6.1.2, some implicit conversions from integral types to `float` or `double` can lose information. That possibility shouldn't be used as an excuse for your own user-defined implicit conversions losing information, though.

A conversion operator declaration that includes the `explicit` keyword introduces a user-defined explicit conversion. Explicit conversions can occur in cast expressions, and are described further in §6.2.

> **BILL WAGNER** In contrast to implicit conversions, explicit conversions can fail, or can lose information, because the user must explicitly ask for the conversion.

A conversion operator converts from a source type, indicated by the parameter type of the conversion operator, to a target type, indicated by the return type of the conversion operator.

For a given source type `S` and target type `T`, if `S` or `T` are nullable types, let S_0 and T_0 refer to their underlying types; otherwise, S_0 and T_0 are equal to `S` and `T`, respectively. A class or struct is permitted to declare a conversion from a source type `S` to a target type `T` only if all of the following are true:

- S_0 and T_0 are different types.
- Either S_0 or T_0 is the class or struct type in which the operator declaration takes place.
- Neither S_0 nor T_0 is an *interface-type*.
- Excluding user-defined conversions, a conversion does not exist from `S` to `T` or from `T` to `S`.

For the purposes of these rules, any type parameters associated with `S` or `T` are considered to be unique types that have no inheritance relationship with other types, and any constraints on those type parameters are ignored.

In the example

```
class C<T> {...}
class D<T>: C<T>

    public static implicit operator C<int>(D<T> value) {...}    // Okay
    public static implicit operator C<string>(D<T> value) {...} // Okay
    public static implicit operator C<T>(D<T> value) {...}      // Error
}
```


www.it-ebooks.info

the first two operator declarations are permitted because, for the purposes of §10.9.3, `T` has no relationship to `int` and `string`, respectively; in both cases, they are considered unique types. However, the third operator is an error because `C<T>` is the base class of `D<T>`.

From the second rule, it follows that a conversion operator must convert either to or from the class or struct type in which the operator is declared. For example, it is possible for a class or struct type `C` to define a conversion from `C` to `int` and from `int` to `C`, but not from `int` to `bool`.

It is not possible to directly redefine a predefined conversion. Thus conversion operators are not allowed to convert from or to `object` because implicit and explicit conversions already exist between `object` and all other types. Likewise, neither the source nor the target types of a conversion can be a base type of the other, since a conversion would then already exist.

However, it *is* possible to declare operators on generic types that, for particular type arguments, specify conversions that already exist as predefined conversions. In the example

```
struct Convertible<T>
{
    public static implicit operator Convertible<T>(T value) {...}

    public static explicit operator T(Convertible<T> value) {...}
}
```

when type `object` is specified as a type argument for `T`, the second operator declares a conversion that already exists (an implicit, and therefore also an explicit, conversion exists from any type to type `object`).

In cases where a predefined conversion exists between two types, any user-defined conversions between those types are ignored. Specifically:

- If a predefined implicit conversion (§6.1) exists from type `S` to type `T`, all user-defined conversions (implicit or explicit) from `S` to `T` are ignored.
- If a predefined explicit conversion (§6.2) exists from type `S` to type `T`, any user-defined explicit conversions from `S` to `T` are ignored. Furthermore:
  - If `T` is an interface type, user-defined implicit conversions from `S` to `T` are ignored.
  - Otherwise, user-defined implicit conversions from `S` to `T` are still considered.

For all types except `object`, the operators declared by the `Convertible<T>` type above do not conflict with predefined conversions. For example:

```
void F(int i, Convertible<int> n) {
    i = n;                      // Error
    i = (int)n;                 // User-defined explicit conversion
    n = i;                      // User-defined implicit conversion
    n = (Convertible<int>)i;    // User-defined implicit conversion
}
```


www.it-ebooks.info

However, for type `object`, predefined conversions hide the user-defined conversions in all cases but one:

```
void F(object o, Convertible<object> n) {
    o = n;                          // Predefined boxing conversion
    o = (object)n;                  // Predefined boxing conversion
    n = o;                          // User-defined implicit conversion
    n = (Convertible<object>)o;     // Predefined unboxing conversion
}
```

User-defined conversions are not allowed to convert from or to *interface-type*s. In particular, this restriction ensures that no user-defined transformations occur when converting to an *interface-type*, and that a conversion to an *interface-type* succeeds only if the object being converted actually implements the specified *interface-type*.

The signature of a conversion operator consists of the source type and the target type. (Note that this is the only form of member for which the return type participates in the signature.) The `implicit` or `explicit` classification of a conversion operator is not part of the operator's signature. Thus a class or struct cannot declare both an `implicit` and an `explicit` conversion operator with the same source and target types.

In general, user-defined implicit conversions should be designed to never throw exceptions and never lose information. If a user-defined conversion can give rise to exceptions (for example, because the source argument is out of range) or loss of information (such as discarding high-order bits), then that conversion should be defined as an explicit conversion.

In the example

```
using System;

public struct Digit
{
    byte value;

    public Digit(byte value) {
        if (value < 0 || value > 9) throw new ArgumentException();
        this.value = value;
    }

    public static implicit operator byte(Digit d) {
        return d.value;
    }

    public static explicit operator Digit(byte b) {
        return new Digit(b);
    }
}
```


www.it-ebooks.info

the conversion from `Digit` to `byte` is implicit because it never throws exceptions or loses information, but the conversion from `byte` to `Digit` is explicit because `Digit` can represent only a subset of the possible values of a `byte`.

10.11 Instance Constructors

An ***instance constructor*** is a member that implements the actions required to initialize an instance of a class. Instance constructors are declared using *constructor-declaration*s:

constructor-declaration:
    attributes$_{opt}$ *constructor-modifiers*$_{opt}$ *constructor-declarator* *constructor-body*

constructor-modifiers:
    constructor-modifier
    constructor-modifiers *constructor-modifier*

constructor-modifier:
    `public`
    `protected`
    `internal`
    `private`
    `extern`

constructor-declarator:
    identifier `(` *formal-parameter-list*$_{opt}$ `)` *constructor-initializer*$_{opt}$

constructor-initializer:
    `:` `base` `(` *argument-list*$_{opt}$ `)`
    `:` `this` `(` *argument-list*$_{opt}$ `)`

constructor-body:
    block
    `;`

A *constructor-declaration* may include a set of *attributes* (§17), a valid combination of the four access modifiers (§10.3.5), and an `extern` (§10.6.7) modifier. A constructor declaration is not permitted to include the same modifier multiple times.

The *identifier* of a *constructor-declarator* must name the class in which the instance constructor is declared. If any other name is specified, a compile-time error occurs.

www.it-ebooks.info

The optional *formal-parameter-list* of an instance constructor is subject to the same rules as the *formal-parameter-list* of a method (§10.6). The formal parameter list defines the signature (§3.6) of an instance constructor and governs the process whereby overload resolution (§7.5.2) selects a particular instance constructor in an invocation.

Each of the types referenced in the *formal-parameter-list* of an instance constructor must be at least as accessible as the constructor itself (§3.5.4).

The optional *constructor-initializer* specifies another instance constructor to invoke before executing the statements given in the *constructor-body* of this instance constructor. This is described further in §10.11.1.

When a constructor declaration includes an `extern` modifier, the constructor is said to be an ***external constructor***. Because an external constructor declaration provides no actual implementation, its *constructor-body* consists of a semicolon. For all other constructors, the *constructor-body* consists of a *block* that specifies the statements to initialize a new instance of the class. This corresponds exactly to the *block* of an instance method with a `void` return type (§10.6.10).

Instance constructors are not inherited. Thus a class has no instance constructors other than those actually declared in the class. If a class contains no instance constructor declarations, a default instance constructor is automatically provided (§10.11.4).

Instance constructors are invoked by *object-creation-expression*s (§7.6.10.1) and through *constructor-initializer*s.

10.11.1 Constructor Initializers

All instance constructors (except those for class `object`) implicitly include an invocation of another instance constructor immediately before the *constructor-body*. The constructor to implicitly invoke is determined by the *constructor-initializer*:

- An instance constructor initializer of the form `base(`*argument-list*$_{opt}$`)` causes an instance constructor from the direct base class to be invoked. That constructor is selected using *argument-list* and the overload resolution rules of §7.5.3. The set of candidate instance constructors consists of all accessible instance constructors contained in the direct base class, or the default constructor (§10.11.4) if no instance constructors are declared in the direct base class. If this set is empty, or if a single best instance constructor cannot be identified, a compile-time error occurs.
- An instance constructor initializer of the form `this(`*argument-list*$_{opt}$`)` causes an instance constructor from the class itself to be invoked. The constructor is selected using *argument-list* and the overload resolution rules of §7.5.3. The set of candidate instance

www.it-ebooks.info

constructors consists of all accessible instance constructors declared in the class itself. If this set is empty, or if a single best instance constructor cannot be identified, a compile-time error occurs. If an instance constructor declaration includes a constructor initializer that invokes the constructor itself, a compile-time error occurs.

> **VLADIMIR RESHETNIKOV** An argument in a *constructor-initializer* can be of type `dynamic` only if the argument has a `ref` or `out` modifier; otherwise, a compile-time error occurs (CS1975). Thus a *constructor-initializer* is never dynamically dispatched.

If an instance constructor has no constructor initializer, a constructor initializer of the form `base()` is implicitly provided. Thus an instance constructor declaration of the form

```
C(...) {...}
```

is exactly equivalent to

```
C(...): base() {...}
```

The scope of the parameters given by the *formal-parameter-list* of an instance constructor declaration includes the constructor initializer of that declaration. Thus a constructor initializer is permitted to access the parameters of the constructor. For example:

```
class A
{
    public A(int x, int y) {}
}

class B: A
{
    public B(int x, int y): base(x + y, x - y) {}
}
```

An instance constructor initializer cannot access the instance being created. Therefore it is a compile-time error to reference `this` in an argument expression of the constructor initializer, as is it a compile-time error for an argument expression to reference any instance member through a *simple-name*.

10.11.2 Instance Variable Initializers

When an instance constructor has no constructor initializer, or it has a constructor initializer of the form `base(...)`, that constructor implicitly performs the initializations specified by the *variable-initializer*s of the instance fields declared in its class. This corresponds to a sequence of assignments that are executed immediately upon entry to the constructor and before the implicit invocation of the direct base class constructor. The variable initializers are executed in the textual order in which they appear in the class declaration.

www.it-ebooks.info

10.11.3 Constructor Execution

Variable initializers are transformed into assignment statements, and these assignment statements are executed before the invocation of the base class instance constructor. This ordering ensures that all instance fields are initialized by their variable initializers before any statements that have access to that instance are executed.

Given the example

```
using System;

class A
{
    public A() {
        PrintFields();
    }

    public virtual void PrintFields() {}
}

class B: A
{
    int x = 1;
    int y;

    public B() {
        y = -1;
    }

    public override void PrintFields() {
        Console.WriteLine("x = {0}, y = {1}", x, y);
    }
}
```

when `new B()` is used to create an instance of `B`, the following output is produced:

```
x = 1, y = 0
```

The value of `x` is 1 because the variable initializer is executed before the base class instance constructor is invoked. However, the value of `y` is 0 (the default value of an `int`) because the assignment to `y` is not executed until after the base class constructor returns.

It is useful to think of instance variable initializers and constructor initializers as statements that are automatically inserted before the *constructor-body*. The example

```
using System;
using System.Collections;

class A
{
    int x = 1, y = -1, count;

    public A() {
        count = 0;
    }
```


www.it-ebooks.info

```
    public A(int n) {
        count = n;
    }
}

class B: A
{
    double sqrt2 = Math.Sqrt(2.0);
    ArrayList items = new ArrayList(100);
    int max;

    public B(): this(100) {
        items.Add("default");
    }

    public B(int n): base(n - 1) {
        max = n;
    }
}
```

contains several variable initializers; it also contains constructor initializers of both forms (`base` and `this`). The example corresponds to the code shown below, where each comment indicates an automatically inserted statement (the syntax used for the automatically inserted constructor invocations isn't valid, but merely serves to illustrate the mechanism).

```
using System.Collections;

class A
{
    int x, y, count;

    public A() {
        x = 1;                      // Variable initializer
        y = -1;                     // Variable initializer
        object();                   // Invoke object() constructor
        count = 0;
    }

    public A(int n) {
        x = 1;                      // Variable initializer
        y = -1;                     // Variable initializer
        object();                   // Invoke object() constructor
        count = n;
    }
}

class B: A
{
    double sqrt2;
    ArrayList items;
    int max;

    public B(): this(100) {
        B(100);                     // Invoke B(int) constructor
        items.Add("default");
    }
```


www.it-ebooks.info

```
    public B(int n): base(n - 1) {
        sqrt2 = Math.Sqrt(2.0);     // Variable initializer
        items = new ArrayList(100); // Variable initializer
        A(n - 1);                   // Invoke A(int) constructor
        max = n;
    }
}
```

10.11.4 Default Constructors

If a class contains no instance constructor declarations, a default instance constructor is automatically provided. That default constructor simply invokes the parameterless constructor of the direct base class. If the direct base class does not have an accessible parameterless instance constructor, a compile-time error occurs. If the class is abstract, then the declared accessibility for the default constructor is `protected`. Otherwise, the declared accessibility for the default constructor is `public`. Thus the default constructor is always of the form

```
protected C(): base() {}
```

or

```
public C(): base() {}
```

where `C` is the name of the class.

In the example

```
class Message
{
    object sender;
    string text;
}
```

a default constructor is provided because the class contains no instance constructor declarations. Thus the example is precisely equivalent to

```
class Message
{
    object sender;
    string text;

    public Message(): base() {}
}
```

10.11.5 Private Constructors

When a class `T` declares only `private` instance constructors, it is not possible for classes outside the program text of `T` to derive from `T` or to directly create instances of `T`. Thus, if a

www.it-ebooks.info

class contains only `static` members and isn't intended to be instantiated, adding an empty `private` instance constructor will prevent instantiation. For example:

```
public class Trig
{
    private Trig() {}       // Prevent instantiation

    public const double PI = 3.14159265358979323846;

    public static double Sin(double x) {...}
    public static double Cos(double x) {...}
    public static double Tan(double x) {...}
}
```

The `Trig` class groups related methods and constants, but is not intended to be instantiated. Therefore it declares a single empty `private` instance constructor. At least one instance constructor must be declared to suppress the automatic generation of a default constructor.

BILL WAGNER If a class is not meant to be instantiated, it would be better to make it a `static` class. However, if you are implementing the singleton pattern, a `private` constructor ensures that only your factory method can create the singleton object.

JOSEPH ALBAHARI If the goal is simply to prevent instantiation, an easier option is to declare the class as `static`.

Private constructors also have a subtler purpose, which relies on the fact that a private constructor can still be invoked from static members of the same class. Sometimes exposing static methods as the only public means of instantiating a class offers certain advantages. For example, with immutable objects, this approach can be used to implement a transparent object caching system.

10.11.6 Optional Instance Constructor Parameters

The `this(...)` form of constructor initializer is commonly used in conjunction with overloading to implement optional instance constructor parameters. In the example

```
class Text
{
    public Text(): this(0, 0, null) {}

    public Text(int x, int y): this(x, y, null) {}

    public Text(int x, int y, string s) {
        // Actual constructor implementation
    }
}
```


www.it-ebooks.info

the first two instance constructors merely provide the default values for the missing arguments. Both use a `this(...)` constructor initializer to invoke the third instance constructor, which actually does the work of initializing the new instance. The effect is that of optional constructor parameters:

```
Text t1 = new Text();              // Same as Text(0, 0, null)
Text t2 = new Text(5, 10);         // Same as Text(5, 10, null)
Text t3 = new Text(5, 20, "Hello");
```

10.12 Static Constructors

A ***static constructor*** is a member that implements the actions required to initialize a closed class type. Static constructors are declared using *static-constructor-declaration*s:

static-constructor-declaration:
attributes$_{opt}$ *static-constructor-modifiers identifier* ( ) *static-constructor-body*

static-constructor-modifiers:
`extern`$_{opt}$ `static`
`static` `extern`$_{opt}$

static-constructor-body:
block
`;`

A *static-constructor-declaration* may include a set of *attributes* (§17) and an `extern` modifier (§10.6.7).

The *identifier* of a *static-constructor-declaration* must name the class in which the static constructor is declared. If any other name is specified, a compile-time error occurs.

When a static constructor declaration includes an `extern` modifier, the static constructor is said to be an ***external static constructor***. Because an external static constructor declaration provides no actual implementation, its *static-constructor-body* consists of a semicolon. For all other static constructor declarations, the *static-constructor-body* consists of a *block* that specifies the statements to execute to initialize the class. This corresponds exactly to the *method-body* of a static method with a `void` return type (§10.6.10).

Static constructors are not inherited, and cannot be called directly.

The static constructor for a closed class type executes at most once in a given application domain. The execution of a static constructor is triggered by the first of the following events to occur within an application domain:

www.it-ebooks.info

- An instance of the class type is created.
- Any of the static members of the class type are referenced.

> **JON SKEET** It's very possible for the presence of an empty static constructor to affect the behavior of the code, as it can change the point at which static field initializers are executed. If you choose to take advantage of this behavior, I would highly recommend that you at least include a comment in the static constructor to explain why and how you're relying on its presence, to avoid it being removed by an eager maintainer at a later date.

If a class contains the `Main` method (§3.1) in which execution begins, the static constructor for that class executes before the `Main` method is called.

To initialize a new closed class type, first a new set of static fields (§10.5.1) for that particular closed type is created. Each of the static fields is initialized to its default value (§5.2). Next, the static field initializers (§10.4.5.1) are executed for those static fields. Finally, the static constructor is executed.

The example

```
using System;

class Test
{
    static void Main()
    {
        A.F();
        B.F();
    }
}

class A
{
    static A()
    {
        Console.WriteLine("Init A");
    }
    public static void F()
    {
        Console.WriteLine("A.F");
    }
}

class B
{
    static B()
    {
        Console.WriteLine("Init B");
    }
```


www.it-ebooks.info

```
    public static void F()
    {
        Console.WriteLine("B.F");
    }
}
```

must produce the output

```
Init A
A.F
Init B
B.F
```

because the execution of A's static constructor is triggered by the call to `A.F`, and the execution of B's static constructor is triggered by the call to `B.F`.

It is possible to construct circular dependencies that allow static fields with variable initializers to be observed in their default value state.

The example

```
using System;

class A
{
    public static int X;

    static A()
    {
        X = B.Y + 1;
    }
}

class B
{
    public static int Y = A.X + 1;

    static B() { }

    static void Main()
    {
        Console.WriteLine("X = {0}, Y = {1}", A.X, B.Y);
    }
}
```

produces the following output:

```
X = 1, Y = 2
```

To execute the `Main` method, the system first runs the initializer for `B.Y`, prior to class B's static constructor. Y's initializer causes A's static constructor to be run because the value of `A.X` is referenced. The static constructor of A, in turn, proceeds to compute the value of X, and in doing so fetches the default value of Y, which is zero. `A.X` is thus initialized to 1. The

www.it-ebooks.info

process of running A's static field initializers and static constructor then completes, returning to the calculation of the initial value of Y, the result of which becomes 2.

Because the static constructor is executed exactly once for each closed constructed class type, it is a convenient place to enforce runtime checks on the type parameter that cannot be checked at compile-time via constraints (§10.1.5). For example, the following type uses a static constructor to enforce that the type argument is an enum:

```
class Gen<T> where T : struct
{
    static Gen()
    {
        if (!typeof(T).IsEnum)
        {
            throw new ArgumentException("T must be an enum");
        }
    }
}
```


10.13 Destructors

A ***destructor*** is a member that implements the actions required to destruct an instance of a class. A destructor is declared using a *destructor-declaration*:

destructor-declaration:
    attributes$_{opt}$ `extern`$_{opt}$ `~` *identifier* `(` `)` *destructor-body*

destructor-body:
    block
    `;`

A *destructor-declaration* may include a set of *attributes* (§17).

The *identifier* of a *destructor-declarator* must name the class in which the destructor is declared. If any other name is specified, a compile-time error occurs.

When a destructor declaration includes an `extern` modifier, the destructor is said to be an ***external destructor***. Because an external destructor declaration provides no actual implementation, its *destructor-body* consists of a semicolon. For all other destructors, the *destructor-body* consists of a *block* that specifies the statements to execute to destruct an instance of the class. A *destructor-body* corresponds exactly to the *method-body* of an instance method with a `void` return type (§10.6.10).

Destructors are not inherited. Thus a class has no destructors other than the one which may be declared in that class.

www.it-ebooks.info

Since a destructor is required to have no parameters, it cannot be overloaded, so a class can have, at most, one destructor.

Destructors are invoked automatically, and cannot be invoked explicitly. An instance becomes eligible for destruction when it is no longer possible for any code to use that instance. Execution of the destructor for the instance may occur at any time after the instance becomes eligible for destruction. When an instance is destructed, the destructors in that instance's inheritance chain are called, in order, from most derived to least derived. A destructor may be executed on any thread. For further discussion of the rules that govern when and how a destructor is executed, see §3.9.

ERIC LIPPERT Clearly, code running in a destructor is running in a potentially very different environment than code anywhere else in your program. It is a really bad idea to do anything complicated, dangerous, side-effecting, or lengthy. In particular, the following code, which produces a complicated side effect of writing to the console is for pedagogic purposes, is *not* an example of what you should do in a real destructor.

The example

```
using System;

class A
{
    ~A()
    {
        Console.WriteLine("A's destructor");
    }
}

class B : A
{
    ~B()
    {
        Console.WriteLine("B's destructor");
    }
}

class Test
{
    static void Main()
    {
        B b = new B();
        b = null;
        GC.Collect();
        GC.WaitForPendingFinalizers();
    }
}
```


www.it-ebooks.info

produces the output

```
B's destructor
A's destructor
```

since destructors in an inheritance chain are called in order, from most derived to least derived.

> **ERIC LIPPERT** Traditionally, the term "destructor" refers to a deterministic cleanup method and a "finalizer" refers to a nondeterministic cleanup method called by a garbage collector. "Destructor" is a bit of an unfortunate misnomer in C#; ideally these methods would be called "finalizers" and the `Dispose` method of `IDisposable` would be called a "destructor."

Destructors are implemented by overriding the virtual method `Finalize` on `System.Object`. C# programs are not permitted to override this method or call it (or overrides of it) directly. For instance, the program

```
class A
{
    override protected void Finalize() { }      // Error

    public void F()
    {
        this.Finalize();                        // Error
    }
}
```

contains two errors.

The compiler behaves as if this method, and overrides of it, do not exist at all. Thus the program

```
class A
{
    void Finalize() { }                         // Permitted
}
```

is valid, and the method shown hides `System.Object`'s `Finalize` method.

For a discussion of the behavior when an exception is thrown from a destructor, see §16.3.

www.it-ebooks.info

10.14 Iterators

A function member (§7.5) implemented using an iterator block (§8.2) is called an ***iterator***.

An iterator block may be used as the body of a function member as long as the return type of the corresponding function member is one of the enumerator interfaces (§10.14.1) or one of the enumerable interfaces (§10.14.2). It can occur as a *method-body*, *operator-body*, or *accessor-body*, whereas events, instance constructors, static constructors, and destructors cannot be implemented as iterators.

When a function member is implemented using an iterator block, it is a compile-time error for the formal parameter list of the function member to specify any `ref` or `out` parameters.

10.14.1 Enumerator Interfaces

The ***enumerator interfaces*** are the nongeneric interface `System.Collections.IEnumerator` and all instantiations of the generic interface `System.Collections.Generic.IEnumerator<T>`. For the sake of brevity, in this chapter these interfaces are referenced as `IEnumerator` and `IEnumerator<T>`, respectively.

> **JON SKEET** I wish the "enumerable" and "enumerator" terminology had been "iterable" and "iterator," including `IIterable<T>` and `IIterator<T>` interfaces. Admittedly, the double-I part is ugly, but it would have kept more lexical space between iterators and enumerations. It's bizarrely long-winded to enumerate over an enumeration.

10.14.2 Enumerable Interfaces

The ***enumerable interfaces*** are the nongeneric interface `System.Collections.IEnumerable` and all instantiations of the generic interface `System.Collections.Generic.IEnumerable<T>`. For the sake of brevity, in this chapter these interfaces are referenced as `IEnumerable` and `IEnumerable<T>`, respectively.

10.14.3 Yield Type

An iterator produces a sequence of values, all of the same type. This type is called the ***yield type*** of the iterator.

- The yield type of an iterator that returns `IEnumerator` or `IEnumerable` is `object`.
- The yield type of an iterator that returns `IEnumerator<T>` or `IEnumerable<T>` is `T`.

www.it-ebooks.info

10.14.4 Enumerator Objects

When a function member returning an enumerator interface type is implemented using an iterator block, invoking the function member does not immediately execute the code in the iterator block. Instead, an ***enumerator object*** is created and returned. This object encapsulates the code specified in the iterator block, and execution of the code in the iterator block occurs when the enumerator object's `MoveNext` method is invoked. An enumerator object has the following characteristics:

- It implements `IEnumerator` and `IEnumerator<T>`, where `T` is the yield type of the iterator.
- It implements `System.IDisposable`.
- It is initialized with a copy of the argument values (if any) and instance value passed to the function member.
- It has four potential states—***before***, ***running***, ***suspended***, and ***after***—and is initially in the ***before*** state.

> **JON SKEET** The fact that *none* of the code within an iterator block runs until the first call to `MoveNext()` is irritating. It means that if you need to check any parameter values, you should really use two methods: a normal method that performs the appropriate validation and calls the second method, which is then implemented with an iterator block. Ideally, some construct would be available to indicate a section of the iterator block that should be executed immediately, before constructing the state machine.

An enumerator object is typically an instance of a compiler-generated enumerator class that encapsulates the code in the iterator block and implements the enumerator interfaces, but other methods of implementation are possible. If an enumerator class is generated by the compiler, that class will be nested, directly or indirectly, in the class containing the function member; it will have private accessibility; and it will have a name reserved for compiler use (§2.4.2).

An enumerator object may implement more interfaces than those specified above.

The following sections describe the exact behavior of the `MoveNext`, `Current`, and `Dispose` members of the `IEnumerable` and `IEnumerable<T>` interface implementations provided by an enumerator object.

Note that enumerator objects do not support the `IEnumerator.Reset` method. Invoking this method causes a `System.NotSupportedException` to be thrown.

www.it-ebooks.info

10.14.4.1 *The MoveNext Method*

The `MoveNext` method of an enumerator object encapsulates the code of an iterator block. Invoking the `MoveNext` method executes code in the iterator block and sets the `Current` property of the enumerator object as appropriate. The precise action performed by `MoveNext` depends on the state of the enumerator object when `MoveNext` is invoked:

- If the state of the enumerator object is ***before***, invoking `MoveNext`:
  - Changes the state to ***running***.
  - Initializes the parameters (including `this`) of the iterator block to the argument values and instance value saved when the enumerator object was initialized.
  - Executes the iterator block from the beginning until execution is interrupted (as described below).
- If the state of the enumerator object is ***running***, the result of invoking `MoveNext` is unspecified.
- If the state of the enumerator object is ***suspended***, invoking `MoveNext`:
  - Changes the state to ***running***.
  - Restores the values of all local variables and parameters (including `this`) to the values saved when execution of the iterator block was last suspended. Note that the contents of any objects referenced by these variables may have changed since the previous call to `MoveNext`.
  - Resumes execution of the iterator block immediately following the `yield return` statement that caused the suspension of execution and continues until execution is interrupted (as described below).
- If the state of the enumerator object is ***after***, invoking `MoveNext` returns `false`.

When `MoveNext` executes the iterator block, execution can be interrupted in four ways: by a `yield return` statement, by a `yield break` statement, by encountering the end of the iterator block, and by an exception being thrown and propagated out of the iterator block.

- When a `yield return` statement is encountered (§8.14):
  - The expression given in the statement is evaluated, implicitly converted to the yield type, and assigned to the `Current` property of the enumerator object.
  - Execution of the iterator body is suspended. The values of all local variables and parameters (including `this`) are saved, as is the location of this `yield return` statement.

www.it-ebooks.info

If the `yield return` statement is within one or more `try` blocks, the associated `finally` blocks are *not* executed at this time.

- The state of the enumerator object is changed to ***suspended***.
- The `MoveNext` method returns `true` to its caller, indicating that the iteration successfully advanced to the next value.

- When a `yield break` statement is encountered (§8.14):
  - If the `yield break` statement is within one or more `try` blocks, the associated `finally` blocks are executed.
  - The state of the enumerator object is changed to ***after***.
  - The `MoveNext` method returns `false` to its caller, indicating that the iteration is complete.
- When the end of the iterator body is encountered:
  - The state of the enumerator object is changed to ***after***.
  - The `MoveNext` method returns `false` to its caller, indicating that the iteration is complete.
- When an exception is thrown and propagated out of the iterator block:
  - Appropriate `finally` blocks in the iterator body will have been executed by the exception propagation.
  - The state of the enumerator object is changed to ***after***.
  - The exception propagation continues to the caller of the `MoveNext` method.

10.14.4.2 *The Current Property*

An enumerator object's `Current` property is affected by `yield return` statements in the iterator block.

When an enumerator object is in the ***suspended*** state, the value of `Current` is the value set by the previous call to `MoveNext`. When an enumerator object is in the ***before***, ***running***, or ***after*** states, the result of accessing `Current` is unspecified.

For an iterator with a yield type other than `object`, the result of accessing `Current` through the enumerator object's `IEnumerable` implementation corresponds to accessing `Current` through the enumerator object's `IEnumerator<T>` implementation and casting the result to `object`.

www.it-ebooks.info

10.14.4.3 *The Dispose Method*

The `Dispose` method is used to clean up the iteration by bringing the enumerator object to the ***after*** state.

- If the state of the enumerator object is ***before***, invoking `Dispose` changes the state to ***after***.
- If the state of the enumerator object is ***running***, the result of invoking `Dispose` is unspecified.
- If the state of the enumerator object is ***suspended***, invoking `Dispose`:
  - Changes the state to ***running***.
  - Executes any `finally` blocks as if the last executed `yield return` statement were a `yield break` statement. If this causes an exception to be thrown and propagated out of the iterator body, the state of the enumerator object is set to ***after*** and the exception is propagated to the caller of the `Dispose` method.
  - Changes the state to ***after***.
- If the state of the enumerator object is ***after***, invoking `Dispose` has no effect.

10.14.5 Enumerable Objects

When a function member returning an enumerable interface type is implemented using an iterator block, invoking the function member does not immediately execute the code in the iterator block. Instead, an ***enumerable object*** is created and returned. The enumerable object's `GetEnumerator` method returns an enumerator object that encapsulates the code specified in the iterator block, and execution of the code in the iterator block occurs when the enumerator object's `MoveNext` method is invoked. An enumerable object has the following characteristics:

- It implements `IEnumerable` and `IEnumerable<T>`, where `T` is the yield type of the iterator.
- It is initialized with a copy of the argument values (if any) and instance value passed to the function member.

An enumerable object is typically an instance of a compiler-generated enumerable class that encapsulates the code in the iterator block and implements the enumerable interfaces, but other methods of implementation are possible. If an enumerable class is generated by the compiler, that class will be nested, directly or indirectly, in the class containing the function member, it will have `private` accessibility, and it will have a name reserved for compiler use (§2.4.2).

www.it-ebooks.info

An enumerable object may implement more interfaces than those specified above. In particular, an enumerable object may also implement `IEnumerator` and `IEnumerator<T>`, enabling it to serve as both an enumerable and an enumerator. In that type of implementation, the first time an enumerable object's `GetEnumerator` method is invoked, the enumerable object itself is returned. Subsequent invocations of the enumerable object's `GetEnumerator`, if any, return a copy of the enumerable object. Thus each returned enumerator has its own state and changes in one enumerator will not affect another.

10.14.5.1 *The GetEnumerator Method*

An enumerable object provides an implementation of the `GetEnumerator` methods of the `IEnumerable` and `IEnumerable<T>` interfaces. The two `GetEnumerator` methods share a common implementation that acquires and returns an available enumerator object. The enumerator object is initialized with the argument values and instance value saved when the enumerable object was initialized, but otherwise the enumerator object functions as described in §10.14.4.

10.14.6 Implementation Example

This section describes a possible implementation of iterators in terms of standard C# constructs. The implementation described here is based on the same principles used by the Microsoft C# compiler, but it is by no means a mandated implementation or the only one possible.

> **BILL WAGNER** As you look at this example, notice that the compiler is simply creating all the code you would generate if you were to create your own nested enumerator class. The iterator method saves you from a great deal of repeated work, and it improves the readability of your code.

The following `Stack<T>` class implements its `GetEnumerator` method using an iterator. The iterator enumerates the elements of the stack in top to bottom order.

```
using System;
using System.Collections;
using System.Collections.Generic;

class Stack<T> : IEnumerable<T>
{
    T[] items;
    int count;
```


www.it-ebooks.info

```
    public void Push(T item)
    {
        if (items == null)
        {
            items = new T[4];
        }
        else if (items.Length == count)
        {
            T[] newItems = new T[count * 2];
            Array.Copy(items, 0, newItems, 0, count);
            items = newItems;
        }
        items[count++] = item;
    }

    public T Pop()
    {
        T result = items[--count];
        items[count] = default(T);
        return result;
    }

    public IEnumerator<T> GetEnumerator()
    {
        for (int i = count - 1; i >= 0; --i) yield return items[i];
    }
}
```


The `GetEnumerator` method can be translated into an instantiation of a compiler-generated enumerator class that encapsulates the code in the iterator block, as shown in the following example:

```
class Stack<T>: IEnumerable<T>
{
    ...

    public IEnumerator<T> GetEnumerator() {
        return new __Enumerator1(this);
    }

    class __Enumerator1: IEnumerator<T>, IEnumerator
    {
        int __state;
        T __current;
        Stack<T> __this;
        int i;

        public __Enumerator1(Stack<T> __this) {
            this.__this = __this;
        }

        public T Current {
            get { return __current; }
        }
```


www.it-ebooks.info

```
        object IEnumerator.Current {
            get { return __current; }
        }

        public bool MoveNext() {
            switch (__state) {
                case 1: goto __state1;
                case 2: goto __state2;
            }
            i = __this.count - 1;
        __loop:
            if (i < 0) goto __state2;
            __current = __this.items[i];
            __state = 1;
            return true;
        __state1:
            --i;
            goto __loop;
        __state2:
            __state = 2;
            return false;
        }

        public void Dispose() {
            __state = 2;
        }

        void IEnumerator.Reset() {
            throw new NotSupportedException();
        }
    }
}
```

In the preceding translation, the code in the iterator block is turned into a state machine and placed in the `MoveNext` method of the enumerator class. Furthermore, the local variable `i` is turned into a field in the enumerator object so it can continue to exist across invocations of `MoveNext`.

The following example prints a simple multiplication table of the integers 1 through 10. The `FromTo` method in the example returns an enumerable object and is implemented using an iterator.

```
using System;
using System.Collections.Generic;

class Test
{
    static IEnumerable<int> FromTo(int from, int to)
    {
        while (from <= to) yield return from++;
    }
```


www.it-ebooks.info

```
    static void Main()
    {
        IEnumerable<int> e = FromTo(1, 10);
        foreach (int x in e)
        {
            foreach (int y in e)
            {
                Console.Write("{0,3} ", x * y);
            }
            Console.WriteLine();
        }
    }
}
```


The `FromTo` method can be translated into an instantiation of a compiler-generated enumerable class that encapsulates the code in the iterator block, as shown in the following example:

```
using System;
using System.Threading;
using System.Collections;
using System.Collections.Generic;

class Test
{
    ...

    static IEnumerable<int> FromTo(int from, int to) {
        return new __Enumerable1(from, to);
    }

    class __Enumerable1:
        IEnumerable<int>, IEnumerable,
        IEnumerator<int>, IEnumerator
    {
        int __state;
        int __current;
        int __from;
        int from;
        int to;
        int i;

        public __Enumerable1(int __from, int to) {
            this.__from = __from;
            this.to = to;
        }

        public IEnumerator<int> GetEnumerator() {
            __Enumerable1 result = this;
            if (Interlocked.CompareExchange(ref __state, 1, 0) != 0) {
                result = new __Enumerable1(__from, to);
                result.__state = 1;
            }
            result.from = result.__from;
            return result;
        }
```


www.it-ebooks.info

```
        IEnumerator IEnumerable.GetEnumerator() {
            return (IEnumerator)GetEnumerator();
        }

        public int Current {
                get { return __current; }
            }

            object IEnumerator.Current {
                get { return __current; }
            }

            public bool MoveNext() {
            switch (__state) {
            case 1:
                if (from > to) goto case 2;
                __current = from++;
                __state = 1;
                return true;
            case 2:
                __state = 2;
                return false;
            default:
                throw new InvalidOperationException();
            }
        }

        public void Dispose() {
            __state = 2;
        }

        void IEnumerator.Reset() {
            throw new NotSupportedException();
        }
    }
}
```

The enumerable class implements both the enumerable interfaces and the enumerator interfaces, enabling it to serve as both an enumerable and an enumerator. The first time the `GetEnumerator` method is invoked, the enumerable object itself is returned. Subsequent invocations of the enumerable object's `GetEnumerator`, if any, return a copy of the enumerable object. Thus each returned enumerator has its own state and changes in one enumerator will not affect another. The `Interlocked.CompareExchange` method is used to ensure thread-safe operation.

The `from` and `to` parameters are turned into fields in the enumerable class. Because `from` is modified in the iterator block, an additional `__from` field is introduced to hold the initial value given to `from` in each enumerator.

The `MoveNext` method throws an `InvalidOperationException` if it is called when `__state` is `0`. This protects against use of the enumerable object as an enumerator object without first calling `GetEnumerator`.

www.it-ebooks.info

The following example shows a simple tree class. The `Tree<T>` class implements its `GetEnumerator` method using an iterator. The iterator enumerates the elements of the tree in infix order.

```
using System;
using System.Collections.Generic;

class Tree<T> : IEnumerable<T>
{
    T value;
    Tree<T> left;
    Tree<T> right;

    public Tree(T value, Tree<T> left, Tree<T> right)
    {
        this.value = value;
        this.left = left;
        this.right = right;
    }

    public IEnumerator<T> GetEnumerator() {
            if (left != null) foreach (T x in left) yield x;
            yield value;
            if (right != null) foreach (T x in right) yield x;
      }
}

class Program
{
    static Tree<T> MakeTree<T>(T[] items, int left, int right)
    {
        if (left > right) return null;
        int i = (left + right) / 2;
        return new Tree<T>(items[i],
            MakeTree(items, left, i - 1),
            MakeTree(items, i + 1, right));
    }

    static Tree<T> MakeTree<T>(params T[] items)
    {
        return MakeTree(items, 0, items.Length - 1);
    }

    // The output of the program is:
    // 1 2 3 4 5 6 7 8 9
    // Mon Tue Wed Thu Fri Sat Sun

    static void Main()
    {
        Tree<int> ints = MakeTree(1, 2, 3, 4, 5, 6, 7, 8, 9);
        foreach (int i in ints) Console.Write("{0} ", i);
        Console.WriteLine();
```


www.it-ebooks.info

```
        Tree<string> strings = MakeTree(
            "Mon", "Tue", "Wed", "Thu", "Fri", "Sat", "Sun");
        foreach (string s in strings) Console.Write("{0} ", s);
        Console.WriteLine();
    }
}
```

The `GetEnumerator` method can be translated into an instantiation of a compiler-generated enumerator class that encapsulates the code in the iterator block, as shown in the following example:

```
class Tree<T>: IEnumerable<T>
{
    ...

    public IEnumerator<T> GetEnumerator() {
        return new __Enumerator1(this);
    }

    class __Enumerator1 : IEnumerator<T>, IEnumerator
    {
        Node<T> __this;
        IEnumerator<T> __left, __right;
        int __state;
        T __current;

        public __Enumerator1(Node<T> __this) {
            this.__this = __this;
        }

        public T Current {
            get { return __current; }
        }

        object IEnumerator.Current {
            get { return __current; }
        }

        public bool MoveNext() {
            try {
                switch (__state) {

                case 0:
                    __state = -1;
                    if (__this.left == null)
                        goto __yield_value;
                    __left = __this.left.GetEnumerator();
                    goto case 1;

                case 1:
                    __state = -2;
                    if (!__left.MoveNext())
                        goto __left_dispose;
                    __current = __left.Current;
                    __state = 1;
                    return true;
```


www.it-ebooks.info

```
            __left_dispose:
                __state = -1;
                __left.Dispose();

            __yield_value:
                __current = __this.value;
                __state = 2;
                return true;

            case 2:
                __state = -1;
                if (__this.right == null) goto __end;
                __right = __this.right.GetEnumerator();
                goto case 3;

            case 3:
                __state = -3;
                if (!__right.MoveNext()) goto __right_dispose;
                __current = __right.Current;
                __state = 3;
                return true;

            __right_dispose:
                __state = -1;
                __right.Dispose();

            __end:
                __state = 4;
                break;

            }
        }
        finally {
            if (__state < 0) Dispose();
        }
        return false;
    }

    public void Dispose() {
        try {
            switch (__state) {

            case 1:
            case -2:
                __left.Dispose();
                break;

            case 3:
            case -3:
                __right.Dispose();
                break;

            }
        }
        finally {
            __state = 4;
        }
    }
```


www.it-ebooks.info

```
            void IEnumerator.Reset() {
                throw new NotSupportedException();
            }
        }
    }
```

The compiler generated temporaries used in the `foreach` statements are lifted into the `__left` and `__right` fields of the enumerator object. The `__state` field of the enumerator object is carefully updated so that the `Dispose()` method will be called correctly if an exception is thrown. Note that it is not possible to write the translated code with simple `foreach` statements.

www.it-ebooks.info

This page intentionally left blank

www.it-ebooks.info

11. Structs

Structs are similar to classes in that they represent data structures that can contain data members and function members. However, unlike classes, structs are value types and do not require heap allocation. A variable of a struct type directly contains the data of the struct, whereas a variable of a class type contains a reference to the data, the latter known as an object.

> **ERIC LIPPERT** The statement that "structs do not require heap allocation" is not the statement "all instances of all structs are always allocated on the stack." First, the second statement is not true: The memory for a `DateTime` field of a `Customer` class will be allocated on the heap along with the rest of the memory of the `Customer` class. Second, whether a local variable of value type is allocated by changing the stack register in the CPU is an implementation detail of a particular version of the framework. The specification is pointing out the opportunity for an optimization here, not stating that a particular allocation pattern is required.

Structs are particularly useful for small data structures that have value semantics. Complex numbers, points in a coordinate system, or key–value pairs in a dictionary are all good examples of structs. Key to these data structures is that they have few data members, that they do not require use of inheritance or referential identity, and that they can be conveniently implemented using value semantics where assignment copies the value instead of the reference.

As described in §4.1.4, the simple types provided by C#, such as `int`, `double`, and `bool`, are, in fact, all struct types. Just as these predefined types are structs, so it is also possible to use structs and operator overloading to implement new "primitive" types in the C# language. Two examples of such types are given at the end of this chapter (§11.4).

> **BILL WAGNER** All simple types are immutable. Any structs you create should also be immutable.

www.it-ebooks.info

JON SKEET While I have always appreciated the *ability* to create user-defined value types, I've almost never found myself needing to do so. When it's appropriate, though it can be extremely useful. For example, I'm currently involved in porting a date/time API from Java, and we have three different value types all wrapping just a `long`. In the Java code, these are often represented as plain `long` values for efficiency—but using different types with different operations (and operators) available has made the C# code *vastly* more readable.

11.1 Struct Declarations

A *struct-declaration* is a *type-declaration* (§9.6) that declares a new struct:

struct-declaration:
attributes$_{opt}$ *struct-modifiers*$_{opt}$ `partial`$_{opt}$ `struct` *identifier* *type-parameter-list*$_{opt}$
struct-interfaces$_{opt}$ *type-parameter-constraints-clauses*$_{opt}$ *struct-body* `;`$_{opt}$

A *struct-declaration* consists of an optional set of *attributes* (§17), followed by an optional set of *struct-modifiers* (§11.1.1), followed by an optional `partial` modifier, followed by the keyword `struct` and an *identifier* that names the struct, followed by an optional *type-parameter-list* specification (§10.1.3), followed by an optional *struct-interfaces* specification (§11.1.2), followed by an optional *type-parameters-constraints-clauses* specification (§10.1.5), followed by a *struct-body* (§11.1.4), optionally followed by a semicolon.

11.1.1 Struct Modifiers

A *struct-declaration* may optionally include a sequence of struct modifiers:

struct-modifiers:
struct-modifier
struct-modifiers *struct-modifier*

struct-modifier:
`new`
`public`
`protected`
`internal`
`private`

It is a compile-time error for the same modifier to appear multiple times in a struct declaration.

The modifiers of a struct declaration have the same meaning as those of a class declaration (§10.1).

www.it-ebooks.info

11.1.2 partial Modifier

The `partial` modifier indicates that this *struct-declaration* is a partial type declaration. Multiple partial struct declarations with the same name within an enclosing namespace or type declaration combine to form one struct declaration, following the rules specified in §10.2.

11.1.3 Struct Interfaces

A struct declaration may include a *struct-interfaces* specification, in which case the struct is said to directly implement the given interface types.

struct-interfaces:
    `:` *interface-type-list*

Interface implementations are discussed further in §13.4.

11.1.4 Struct Body

The *struct-body* of a struct defines the members of the struct.

struct-body:
    `{` *struct-member-declarations*$_{opt}$ `}`

11.2 Struct Members

The members of a struct consist of the members introduced by its *struct-member-declaration*s and the members inherited from the type `System.ValueType`.

struct-member-declarations:
    struct-member-declaration
    struct-member-declarations *struct-member-declaration*

struct-member-declaration:
    constant-declaration
    field-declaration
    method-declaration
    property-declaration
    event-declaration
    indexer-declaration
    operator-declaration
    constructor-declaration
    static-constructor-declaration
    type-declaration

Except for the differences noted in §11.3, the descriptions of class members provided in §10.3 through §10.14 apply to struct members as well.

www.it-ebooks.info

11.3 Class and Struct Differences

Structs differ from classes in several important ways:

- Structs are value types (§11.3.1).
- All struct types implicitly inherit from the class `System.ValueType` (§11.3.2).
- Assignment to a variable of a struct type creates a *copy* of the value being assigned (§11.3.3).
- The default value of a struct is the value produced by setting all value type fields to their default values and all reference type fields to `null` (§11.3.4).
- Boxing and unboxing operations are used to convert between a struct type and `object` (§11.3.5).
- The meaning of `this` is different for structs (§7.6.7).
- Instance field declarations for a struct are not permitted to include variable initializers (§11.3.7).
- A struct is not permitted to declare a parameterless instance constructor (§11.3.8).
- A struct is not permitted to declare a destructor (§11.3.9).

JESSE LIBERTY Note that these differences are semantically significant, unlike the trivial differences between structs and classes in C++.

11.3.1 Value Semantics

Structs are value types (§4.1) and are said to have value semantics. Classes, in contrast, are reference types (§4.2) and are said to have reference semantics.

BILL WAGNER The statement below that a variable of a struct type directly containing the data of the struct is one of those correct but misleading specification statements. A struct may contain members of reference types. The data of the struct is a reference to the class type. For example:

```
struct Message
{
    int code;
    string message;
    // message contains a reference to a string
    // object, not the characters in the message
}
```

Assignment copies the reference to message, not the characters themselves.

www.it-ebooks.info

A variable of a struct type directly contains the data of the struct, whereas a variable of a class type contains a reference to the data, the latter known as an object. When a struct `B` contains an instance field of type `A` and `A` is a struct type, it is a compile-time error for `A` to depend on `B`. A struct `X` ***directly depends on*** a struct `Y` if `X` contains an instance field of type `Y`. Given this definition, the complete set of structs upon which a struct depends is the transitive closure of the ***directly depends on*** relationship. For example,

```
struct Node
{
    int data;

    Node next; // Error: Node directly depends on itself

}
```

is an error because `Node` contains an instance field of its own type. Another example

```
struct A { B b; }

struct B { C c; }

struct C { A a; }
```

is an error because each of the types `A`, `B`, and `C` depend on each other.

With classes, it is possible for two variables to reference the same object, and thus possible for operations on one variable to affect the object referenced by the other variable. With structs, each of the variables has its own copy of the data (except in the case of `ref` and `out` parameter variables), and it is not possible for operations on one to affect the others. Furthermore, because structs are not reference types, it is not possible for values of a struct type to be `null`.

Given the declaration

```
struct Point
{
    public int x, y;

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

the code fragment

```
Point a = new Point(10, 10);
Point b = a;
a.x = 100;
System.Console.WriteLine(b.x);
```


www.it-ebooks.info

outputs the value `10`. The assignment of `a` to `b` creates a copy of the value, and `b` is thus unaffected by the assignment to `a.x`. Had `Point` instead been declared as a class, the output would be `100` because `a` and `b` would reference the same object.

11.3.2 Inheritance

All struct types implicitly inherit from the class `System.ValueType`, which in turn inherits from class `object`. A struct declaration may specify a list of implemented interfaces, but it is not possible for a struct declaration to specify a base class.

Struct types are never abstract and are always implicitly sealed. The `abstract` and `sealed` modifiers are, therefore, not permitted in a struct declaration.

Since inheritance isn't supported for structs, the declared accessibility of a struct member cannot be `protected` or `protected internal`.

Function members in a struct cannot be `abstract` or `virtual`, and the `override` modifier is allowed only to override methods inherited from `System.ValueType`.

11.3.3 Assignment

Assignment to a variable of a struct type creates a *copy* of the value being assigned. This differs from assignment to a variable of a class type, which copies the reference but not the object identified by the reference.

Similar to an assignment, when a struct is passed as a value parameter or returned as the result of a function member, a copy of the struct is created. A struct may be passed by reference to a function member using a `ref` or `out` parameter.

> **ERIC LIPPERT** Another way of looking at this issue is that `ref` and `out` parameters create an alias to variables. Rather than thinking, "I'm going to use a `ref` parameter to pass this struct by reference," I prefer to think, "I'm going to use a `ref` parameter to make this parameter an alias for the variable that contains this struct."

When a property or indexer of a struct is the target of an assignment, the instance expression associated with the property or indexer access must be classified as a variable. If the instance expression is classified as a value, a compile-time error occurs. This is described in further detail in §7.17.1.

11.3.4 Default Values

As described in §5.2, several kinds of variables are automatically initialized to their default values when they are created. For variables of class types and other reference types, this

www.it-ebooks.info

default value is `null`. However, since structs are value types that cannot be `null`, the default value of a struct is the value produced by setting all value type fields to their default value and all reference type fields to `null`.

Referring to the `Point` struct declared above, the example

```
Point[] a = new Point[100];
```

initializes each `Point` in the array to the value produced by setting the `x` and `y` fields to zero.

The default value of a struct corresponds to the value returned by the default constructor of the struct (§4.1.2). Unlike a class, a struct is not permitted to declare a parameterless instance constructor. Instead, every struct implicitly has a parameterless instance constructor, which always returns the value that results from setting all value type fields to their default value and all reference type fields to `null`.

Structs should be designed to consider the default initialization state a valid state. In the example

```
using System;

struct KeyValuePair
{
    string key;
    string value;

    public KeyValuePair(string key, string value) {
        if (key == null || value == null)
            throw new ArgumentException();
        this.key = key;
        this.value = value;
    }
}
```

the user-defined instance constructor protects against null values only where it is explicitly called. In cases where a `KeyValuePair` variable is subject to default value initialization, the `key` and `value` fields will be null, and the struct must be prepared to handle this state.

11.3.5 Boxing and Unboxing

BILL WAGNER Since the introduction of C# 2.0, you have often been able to avoid boxing and unboxing by using generics.

A value of a class type can be converted to type `object` or to an interface type that is implemented by the class simply by treating the reference as another type at compile time.

www.it-ebooks.info

Likewise, a value of type `object` or a value of an interface type can be converted back to a class type without changing the reference (but, of course, a runtime type check is required in this case).

Since structs are not reference types, these operations are implemented differently for struct types. When a value of a struct type is converted to type `object` or to an interface type that is implemented by the struct, a boxing operation takes place. Likewise, when a value of type `object` or a value of an interface type is converted back to a struct type, an unboxing operation takes place. A key difference from the same operations on class types is that boxing and unboxing *copy* the struct value either into or out of the boxed instance. Thus, following a boxing or unboxing operation, changes made to the unboxed struct are not reflected in the boxed struct.

> **ERIC LIPPERT** This is yet another reason why value types should be immutable: If a change is impossible, then the fact that changes made to an unboxed struct are not reflected in the boxed struct becomes irrelevant. Rather than dealing with the unexpected and confusing semantics, avoid them altogether.

When a struct type overrides a virtual method inherited from `System.Object` (such as `Equals`, `GetHashCode`, or `ToString`), invocation of the virtual method through an instance of the struct type does not cause boxing to occur. This is true even when the struct is used as a type parameter and the invocation occurs through an instance of the type parameter type. For example:

```
using System;

struct Counter
{
    int value;

    public override string ToString() {
        value++;
        return value.ToString();
    }
}

class Program
{
    static void Test<T>() where T: new() {
        T x = new T();
        Console.WriteLine(x.ToString());
        Console.WriteLine(x.ToString());
        Console.WriteLine(x.ToString());
    }

    static void Main() {
        Test<Counter>();
    }
}
```


www.it-ebooks.info

The output of this program is

```
1
2
3
```

Although it is bad style for `ToString` to have side effects, the example demonstrates that no boxing occurred for the three invocations of `x.ToString()`.

Similarly, boxing never implicitly occurs when accessing a member on a constrained type parameter. For example, suppose an interface `ICounter` contains a method `Increment` that can be used to modify a value. If `ICounter` is used as a constraint, the implementation of the `Increment` method is called with a reference to the variable that `Increment` was called on—never a boxed copy.

```
using System;

interface ICounter
{
    void Increment();
}

struct Counter: ICounter
{
    int value;

    public override string ToString() {
        return value.ToString();
    }

    void ICounter.Increment() {
        value++;
    }
}

class Program
{
    static void Test<T>() where T: ICounter, new() {
        T x = new T();
        Console.WriteLine(x);
        x.Increment();
    // Modify x
        Console.WriteLine(x);
        ((ICounter)x).Increment();
    // Modify boxed copy of x
        Console.WriteLine(x);
    }

    static void Main() {
        Test<Counter>();
    }
}
```


www.it-ebooks.info

The first call to `Increment` modifies the value in the variable `x`. This is not equivalent to the second call to `Increment`, which modifies the value in a boxed copy of `x`. Thus the output of the program is

```
0
1
1
```

For further details on boxing and unboxing, see §4.3.

11.3.6 Meaning of this

Within an instance constructor or instance function member of a class, `this` is classified as a value. Thus, while `this` can be used to refer to the instance for which the function member was invoked, it is not possible to assign to `this` in a function member of a class.

Within an instance constructor of a struct, `this` corresponds to an `out` parameter of the struct type; within an instance function member of a struct, `this` corresponds to a `ref` parameter of the struct type. In both cases, `this` is classified as a variable, and it is possible to modify the entire struct for which the function member was invoked by assigning to `this` or by passing this as a `ref` or `out` parameter.

> **VLADIMIR RESHETNIKOV** Anonymous functions and query expressions inside structs cannot access `this` or instance members of `this`.

11.3.7 Field Initializers

As described in §11.3.4, the default value of a struct consists of the value that results from setting all value type fields to their default values and all reference type fields to `null`. For this reason, a struct does not permit instance field declarations to include variable initializers. This restriction applies only to instance fields. Static fields of a struct are permitted to include variable initializers.

The example

```
struct Point
{
    public int x = 1;  // Error: initializer not permitted
    public int y = 1;  // Error: initializer not permitted
}
```

is in error because the instance field declarations include variable initializers.

www.it-ebooks.info

JOSEPH ALBAHARI The following pattern provides a work-around, giving `X` a default value of `1`:

```
struct Point
{
    bool initialized;    // Default value false
    int x;
    public int X {
        get {
            if (!initialized) { x = 1; initialized = true; }
            return x;
        }
    }
}
```

ERIC LIPPERT A more terse version of Joseph's technique would be to hide the flag in a nullable and use the null coalescing operator:

```
struct Point
{
   private int? x; // Default value is null
   public int X { get { return x ?? 1; }}
}
```

Because a nullable int is actually implemented as a struct containing an `int` and a `bool`, this is essentially the same technique, just a bit more concise.

11.3.8 Constructors

Unlike a class, a struct is not permitted to declare a parameterless instance constructor. Instead, every struct implicitly has a parameterless instance constructor, which always returns the value that results from setting all value type fields to their default values and all reference type fields to null (§4.1.2). A struct can declare instance constructors having parameters. For example:

```
struct Point
{
    int x, y;

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


www.it-ebooks.info

Given the above declaration, the statements

```
Point p1 = new Point();
Point p2 = new Point(0, 0);
```

both create a `Point` with `x` and `y` initialized to zero.

A struct instance constructor is not permitted to include a constructor initializer of the form `base(...)`.

If the struct instance constructor doesn't specify a constructor initializer, the `this` variable corresponds to an `out` parameter of the struct type; similar to an `out` parameter, `this` must be definitely assigned (§5.3) at every location where the constructor returns. If the struct instance constructor specifies a constructor initializer, the `this` variable corresponds to a `ref` parameter of the struct type; similar to a `ref` parameter, `this` is considered definitely assigned on entry to the constructor body. Consider the instance constructor implementation below:

```
struct Point
{
    int x, y;

    public int X {
        set { x = value; }
    }

    public int Y {
        set { y = value; }
    }

    public Point(int x, int y) {
        X = x;    // Error: this is not yet definitely assigned
        Y = y;    // Error: this is not yet definitely assigned
    }
}
```

No instance member function (including the `set` accessors for the properties `X` and `Y`) can be called until all fields of the struct being constructed have been definitely assigned. Note, however, that if `Point` were a class instead of a struct, the instance constructor implementation would be permitted.

ERIC LIPPERT The specification does not mention that the way to make this work is to simply force a call to the default constructor to ensure that the fields are initialized:

```
public Point(int x, int y) : this() { // Struct is guaranteed to be initialized
```


www.it-ebooks.info

11.3.9 Destructors

A struct is not permitted to declare a destructor.

11.3.10 Static Constructors

Static constructors for structs follow most of the same rules as for classes. The execution of a static constructor for a struct type is triggered by the first of the following events to occur within an application domain:

- A static member of the struct type is referenced.
- An explicitly declared constructor of the struct type is called.

The creation of default values (§11.3.4) of struct types does not trigger the static constructor. (An example of this is the initial value of elements in an array.)

11.4 Struct Examples

The following shows two significant examples of using `struct` types to create types that can be used similarly to the built-in types of the language, but with modified semantics.

> **BILL WAGNER** These examples are not as compelling now that nullable types have been added to the language and the framework. Even so, I find that I still create `struct` types for holding instances of types when I create very large collections of data values.

11.4.1 Database Integer Type

The `DBInt` struct below implements an integer type that can represent the complete set of values of the `int` type, plus an additional state that indicates an unknown value. A type with these characteristics is commonly used in databases.

```
using System;

public struct DBInt
{
    // The Null member represents an unknown DBInt value.

    public static readonly DBInt Null = new DBInt();

    // When the defined field is true, this DBInt represents a
    // known value that is stored in the value field. When
    // the defined field is false, this DBInt represents an
    // unknown value, and the value field is 0.
```


www.it-ebooks.info

```
int value;
bool defined;

// Private instance constructor.
// Creates a DBInt with a known value.

DBInt(int value) {
    this.value = value;
    this.defined = true;
}

// The IsNull property is true if this
// DBInt represents an unknown value.

public bool IsNull { get { return !defined; } }

// The Value property is the known value of this DBInt,
// or 0 if this DBInt represents an unknown value.

public int Value { get { return value; } }

// Implicit conversion from int to DBInt.

public static implicit operator DBInt(int x) {
    return new DBInt(x);
}

// Explicit conversion from DBInt to int. Throws an
// exception if the given DBInt represents an unknown value.

public static explicit operator int(DBInt x) {
    if (!x.defined) throw new InvalidOperationException();
    return x.value;
}

public static DBInt operator +(DBInt x) {
    return x;
}

public static DBInt operator -( DBInt x) {
    return x.defined ? -x.value : Null;
}

public static DBInt operator +( DBInt x, DBInt y) {
    return x.defined && y.defined? x.value + y.value: Null;
}

public static DBInt operator -( DBInt x, DBInt y) {
    return x.defined && y.defined? x.value - y.value: Null;
}

public static DBInt operator *( DBInt x, DBInt y) {
    return x.defined && y.defined? x.value * y.value: Null;
}

public static DBInt operator /(DBInt x, DBInt y) {
    return x.defined && y.defined? x.value / y.value: Null;
}

public static DBInt operator %(DBInt x, DBInt y) {
    return x.defined && y.defined? x.value % y.value: Null;
}
```


www.it-ebooks.info

```
    public static DBBool operator ==(DBInt x, DBInt y) {
        return x.defined &&
                    y.defined? x.value == y.value: DBBool.Null;
    }

    public static DBBool operator !=(DBInt x, DBInt y) {
        return x.defined &&
                    y.defined? x.value != y.value: DBBool.Null;
    }

    public static DBBool operator >(DBInt x, DBInt y) {
        return x.defined &&
                    y.defined? x.value > y.value: DBBool.Null;
    }

    public static DBBool operator <(DBInt x, DBInt y) {
        return x.defined &&
                    y.defined? x.value < y.value: DBBool.Null;
    }

    public static DBBool operator >=(DBInt x, DBInt y) {
        return x.defined &&
                    y.defined? x.value >= y.value: DBBool.Null;
    }

    public static DBBool operator <=(DBInt x, DBInt y) {
        return x.defined &&
                    y.defined? x.value <= y.value: DBBool.Null;
    }

    public override bool Equals(object obj) {
        if (!(obj is DBInt)) return false;
        DBInt x = (DBInt)obj;
        return value == x.value && defined == x.defined;
    }

    public override int GetHashCode() {
        return value;
    }

    public override string ToString() {
        return defined? value.ToString():"DBInt.Null";
    }
}
```

CHRISTIAN NAGEL In the namespace `System.Data.SqlTypes`, the .NET contains mapping structs such as `SqlBoolean` and `SqlInt32` that are somewhat similar to the examples `DBInt` and `DBBool` here. These types have been available since .NET 1.0 and have been created to allow for a `null` value, as it is possible with the native database types. Given that nullable types have been available since .NET 2.0, creation of custom types can be avoided and `int?` and `bool?` used instead.

www.it-ebooks.info

11.4.2 Database Boolean Type

The `DBBool` struct below implements a three-valued logical type. The possible values of this type are `DBBool.True`, `DBBool.False`, and `DBBool.Null`, where the `Null` member indicates an unknown value. Such three-valued logical types are commonly used in databases.

```
using System;

public struct DBBool
{
    // The three possible DBBool values.

    public static readonly DBBool Null = new DBBool(0);
    public static readonly DBBool False = new DBBool(-1);
    public static readonly DBBool True = new DBBool(1);

    // Private field that stores -1, 0, 1 for False, Null, True.

    sbyte value;

    // Private instance constructor.
    // The value parameter must be -1, 0, or 1.

    DBBool(int value) {
        this.value = (sbyte)value;
    }

    // Properties to examine the value of a DBBool. Return true
    // if this DBBool has the given value, false otherwise.

    public bool IsNull { get { return value == 0; } }

    public bool IsFalse { get { return value < 0; } }

    public bool IsTrue { get { return value > 0; } }

    // Implicit conversion from bool to DBBool. Maps true to
    // DBBool.True and false to DBBool.False.

    public static implicit operator DBBool(bool x) {
        return x? True: False;
    }

    // Explicit conversion from DBBool to bool.
    // Throws an exception if the given DBBool
    // is Null; otherwise, returns true or false.

    public static explicit operator bool(DBBool x) {
        if (x.value == 0) throw new InvalidOperationException();
        return x.value > 0;
    }

    // Equality operator. Returns Null if either operand is Null;
    // otherwise, returns True or False.

    public static DBBool operator ==(DBBool x, DBBool y) {
        if (x.value == 0 || y.value == 0) return Null;
        return x.value == y.value? True: False;
    }
```


www.it-ebooks.info

```
    // Inequality operator. Returns Null if either operand is Null;
    // otherwise, returns True or False.

    public static DBBool operator !=(DBBool x, DBBool y) {
        if (x.value == 0 || y.value == 0) return Null;
        return x.value != y.value? True: False;
    }

    // Logical negation operator. Returns True if the operand is
    // False, Null if the operand is Null,
    // or False if the operand is True.

    public static DBBool operator !(DBBool x) {
        return new DBBool(-x.value);
    }

    // Logical AND operator. Returns False if either operand is False;
    // otherwise, Null if either operand is Null; otherwise, True.

    public static DBBool operator &(DBBool x, DBBool y) {
        return new DBBool(x.value < y.value? x.value: y.value);
    }

    // Logical OR operator. Returns True if either operand is True;
    // otherwise, Null if either operand is Null; otherwise, False.

    public static DBBool operator |(DBBool x, DBBool y) {
        return new DBBool(x.value > y.value? x.value: y.value);
    }

    // Definitely true operator. Returns true if the
    // operand is True, false otherwise.

    public static bool operator true(DBBool x) {
        return x.value > 0;
    }
    // Definitely false operator. Returns true if the
    // operand is False, false otherwise.

    public static bool operator false(DBBool x) {
        return x.value < 0;
    }

    public override bool Equals(object obj) {
        if (!(obj is DBBool)) return false;
        return value == ((DBBool)obj).value;
    }

    public override int GetHashCode() {
        return value;
    }

    public override string ToString() {
        if (value > 0) return "DBBool.True";
        if (value < 0) return "DBBool.False";
        return "DBBool.Null";
    }
}
```


www.it-ebooks.info

CHRIS SELLS When .NET was first introduced, user-defined value types were an important differentiator of this system from Java. In practice, I find these types mostly used as an optimization after profiling has revealed the garbage collector overworking itself toward no worthy goal.

ERIC LIPPERT The fact that some local variables of value type can be cheaply allocated on the stack does not automatically make them "higher performance" than reference types. Heap allocation is somewhat more expensive than stack allocation, but still pretty cheap. The real cost savings typically come through the deallocation: The fewer objects that are allocated on the heap, the less potential garbage the garbage collector has to identify and compact. Even so, any cost savings in allocation and deallocation is often eaten up by the need to copy value types by value. Processors are optimized for copying things whose size approximates the size of a reference. Value types can be of an odd size and copied a lot, which can in some cases increase the total cost.

My advice is to make the value-versus-reference choice based on whether copy-by-value or copy-by-reference makes more sense, then do performance testing with a profiler. Only change a reference type to a value type (or vice versa) if you have good, empirical, repeatable data that indicates that doing so makes a measurable and important difference.

www.it-ebooks.info

12. Arrays

An array is a data structure that contains a number of variables that are accessed through computed indices. The variables contained in an array, also called the elements of the array, are all of the same type, and this type is called the element type of the array.

An array has a rank that determines the number of indices associated with each array element. The rank of an array is also referred to as the dimensions of the array. An array with a rank of 1 is called a ***single-dimensional array***. An array with a rank greater than 1 is called a ***multi-dimensional array***. Specific-sized multi-dimensional arrays are often referred to as two-dimensional arrays, three-dimensional arrays, and so on.

Each dimension of an array has an associated length that is an integral number greater than or equal to zero. The dimension lengths are not part of the type of the array, but rather are established when an instance of the array type is created at runtime. The length of a dimension determines the valid range of indices for that dimension: For a dimension of length `N`, indices can range from `0` to `N - 1` inclusive. The total number of elements in an array is the product of the lengths of each dimension in the array. If one or more of the dimensions of an array have a length of zero, the array is said to be empty.

The element type of an array can be any type, including an array type.

12.1 Array Types

An array type is written as a *non-array-type* followed by one or more *rank-specifier*s:

array-type:
    non-array-type *rank-specifiers*

non-array-type:
    type

rank-specifiers:
    rank-specifier
    rank-specifiers *rank-specifier*

www.it-ebooks.info

rank-specifier:
`[` *dim-separators*$_{opt}$ `]`

dim-separators:
`,`
dim-separators `,`

A *non-array-type* is any *type* that is not itself an *array-type*.

The rank of an array type is given by the leftmost *rank-specifier* in the *array-type*: A *rank-specifier* indicates that the array is an array with a rank of one plus the number of "`,`" tokens in the *rank-specifier*.

The element type of an array type is the type that results from deleting the leftmost *rank-specifier*:

- An array type of the form `T[R]` is an array with rank `R` and a non-array element type `T`.
- An array type of the form `T[R][`R_1`]...[`R_N`]` is an array with rank `R` and an element type `T[`R_1`]...[`R_N`]`.

In effect, the *rank-specifier*s are read from left to right *before* the final non-array element type. The type `int[][,,][,]` is a single-dimensional array of three-dimensional arrays of two-dimensional arrays of `int`.

> **ERIC LIPPERT** The fact that the rank specifiers of arrays-of-arrays are read "backward" is frequently confusing to people. In reality, this scheme has the nice property that the indexing operations go in the same order as the declarations; you would index that complicated array as `arr[a][b,c,d][e,f]`.

At runtime, a value of an array type can be `null` or a reference to an instance of that array type.

12.1.1 The System.Array Type

The type `System.Array` is the abstract base type of all array types. An implicit reference conversion (§6.1.6) exists from any array type to `System.Array`, and an explicit reference conversion (§6.2.4) exists from `System.Array` to any array type. Note that `System.Array` is not itself an *array-type*. Rather, it is a *class-type* from which all *array-type*s are derived.

At runtime, a value of type `System.Array` can be `null` or a reference to an instance of any array type.

www.it-ebooks.info

12.1.2 Arrays and the Generic IList Interface

A one-dimensional array `T[]` implements the interface `System.Collections.Generic.IList<T>` (`IList<T>` for short) and its base interfaces. Accordingly, there is an implicit conversion from `T[]` to `IList<T>` and its base interfaces. In addition, if there is an implicit reference conversion from `S` to `T`, then `S[]` implements `IList<T>` and there is an implicit reference conversion from `S[]` to `IList<T>` and its base interfaces (§6.1.6). If there is an explicit reference conversion from `S` to `T`, then there is an explicit reference conversion from `S[]` to `IList<T>` and its base interfaces (§6.2.4). For example:

```
using System.Collections.Generic;

class Test
{
    static void Main() {
        string[] sa = new string[5];
        object[] oa1 = new object[5];
        object[] oa2 = sa;

        IList<string> lst1 = sa;                  // Okay
        IList<string> lst2 = oa1;                 // Error: cast needed
        IList<object> lst3 = sa;                  // Okay
        IList<object> lst4 = oa1;                 // Okay

        IList<string> lst5 = (IList<string>)oa1;  // Exception
        IList<string> lst6 = (IList<string>)oa2;  // Okay
    }
}
```

The assignment `lst2 = oa1` generates a compile-time error since the conversion from `object[]` to `IList<string>` is an explicit conversion, not implicit. The cast `(IList<string>)oa1` will cause an exception to be thrown at runtime since `oa1` references an `object[]` and not a `string[]`. However, the cast `(IList<string>)oa2` will not cause an exception to be thrown since `oa2` references a `string[]`.

Whenever there is an implicit or explicit reference conversion from `S[]` to `IList<T>`, there is also an explicit reference conversion from `IList<T>` and its base interfaces to `S[]` (§6.2.4).

When an array type `S[]` implements `IList<T>`, some of the members of the implemented interface may throw exceptions. The precise behavior of the implementation of the interface is beyond the scope of this specification.

ERIC LIPPERT It's interesting to consider the counterfactual world in which the CLR had generic types in version 1. In that counterfactual world, there would probably be generic array types `Array<T>`, `Array2<T>`, and so on. In this world, declarations of complicated array types become more clear: An `Array<Array2<int>>` is a one-dimensional array where every element is a two-dimensional array of integers.

www.it-ebooks.info

> **BILL WAGNER** The fact that an array `T[]` implements the `IList<T>` by throwing exceptions for some methods is one of those troubling real-world problems for which there isn't a good solution. `IList<T>` has methods that add or remove elements. Arrays have a fixed size, however, and cannot support some of those methods. Even so, you want to use the random access to elements that exists in the `IList<T>` interface.

12.2 Array Creation

Array instances are created by *array-creation-expression*s (§7.6.10.4) or by field or local variable declarations that include an *array-initializer* (§12.6).

When an array instance is created, the rank and length of each dimension are established and then remain constant for the entire lifetime of the instance. In other words, it is not possible to change the rank of an existing array instance, nor is it possible to resize its dimensions.

An array instance is always of an array type. The `System.Array` type is an abstract type that cannot be instantiated.

Elements of arrays created by *array-creation-expression*s are always initialized to their default values (§5.2).

12.3 Array Element Access

Array elements are accessed using *element-access* expressions (§7.6.6.1) of the form `A[`I_1`, `I_2`, ..., `I_N`]`, where `A` is an expression of an array type and each I_x is an expression of type `int`, `uint`, `long`, or `ulong`, or can be implicitly converted to one or more of these types. The result of an array element access is a variable—namely, the array element selected by the indices.

The elements of an array can be enumerated using a `foreach` statement (§8.8.4).

12.4 Array Members

Every array type inherits the members declared by the `System.Array` type.

www.it-ebooks.info

12.5 Array Covariance

For any two *reference-type*s `A` and `B`, if an implicit reference conversion (§6.1.6) or explicit reference conversion (§6.2.4) exists from `A` to `B`, then the same reference conversion also exists from the array type `A[R]` to the array type `B[R]`, where `R` is any given *rank-specifier* (but the same for both array types). This relationship is known as ***array covariance***. Array covariance, in particular, means that a value of an array type `A[R]` may actually be a reference to an instance of an array type `B[R]`, provided an implicit reference conversion exists from `B` to `A`.

Because of array covariance, assignments to elements of reference type arrays include a runtime check that ensures the value being assigned to the array element is actually of a permitted type (§7.17.1). For example:

```
class Test
{
    static void Fill(object[] array, int index, int count, object value) {
        for (int i = index; i < index + count; i++)
             array[i] = value;
    }

    static void Main() {
        string[] strings = new string[100];
        Fill(strings, 0, 100, "Undefined");
        Fill(strings, 0, 10, null);
        Fill(strings, 90, 10, 0);
    }
}
```

The assignment to `array[i]` in the `Fill` method implicitly includes a runtime check that ensures the object referenced by `value` is either `null` or an instance that is compatible with the actual element type of `array`. In `Main`, the first two invocations of `Fill` succeed, but the third invocation causes a `System.ArrayTypeMismatchException` to be thrown upon executing the first assignment to `array[i]`. The exception occurs because a boxed `int` cannot be stored in a `string` array.

> **ERIC LIPPERT** This is my candidate for "worst feature" of C#. It allows assignments to fail at runtime without any indication in the source code that such failure is possible. It imposes a performance cost on extremely common code to make a rare scenario go quickly; accessing an array of unsealed reference type safely happens much more often than covariant array conversions do. I much prefer the type-safe covariance that has been added to `IEnumerable<T>`.

www.it-ebooks.info

CHRIS SELLS I find myself using arrays so seldomly—even single-dimensional arrays, let alone multi-dimensional or jagged arrays—that the strange corner cases don't tend to matter much. Given the availability of `List<T>`, `Dictionary<K, V>`, and `IEnumerable<T>`, I'd be a happy guy without arrays at all.

PETER SESTOFT The necessity for the runtime check that Eric laments stems from the array type being covariant in the element type (`Student[]` being a subtype of `Person[]` when `Student` is a subtype of `Person`). This array covariance design weakness is shared with the Java programming language, where the situation is even worse: Because Java implements generics by erasure, there is no type object representing a generic type parameter at runtime, so the check cannot be performed if the array was created as `new T[...]` for some type parameter `T`, or as `new Stack<Person>[...]`. Hence Java must forbid the creation of an array whose element type is a type parameter or a type constructed as a type instance of a generic type. In C#, type parameters and constructed types are represented faithfully at runtime, so these restrictions on array creation do not exist.

In the Scala language, which was born with generic types, the array type is invariant in the element type and the runtime check is not needed (but is likely to be performed anyway, when Scala is compiled to Java bytecode).

BILL WAGNER Eric's comments go a long way toward explaining why the safe covariance and contravariance added for generics in C# 4.0 are so important. Improving both speed and correctness with the same change is generally rare.

Array covariance specifically does not extend to arrays of *value-type*s. For example, no conversion exists that permits an `int[]` to be treated as an `object[]`.

12.6 Array Initializers

Array initializers may be specified in field declarations (§10.5), local variable declarations (§8.5.1), and array creation expressions (§7.6.10.4):

array-initializer:
`{` *variable-initializer-list*$_{opt}$ `}`
`{` *variable-initializer-list* `,` `}`

www.it-ebooks.info

variable-initializer-list:
    variable-initializer
    variable-initializer-list , *variable-initializer*

variable-initializer:
    expression
    array-initializer

An array initializer consists of a sequence of variable initializers, enclosed by "{"and "}" tokens and separated by "," tokens. Each variable initializer is an expression or, in the case of a multi-dimensional array, a nested array initializer.

The context in which an array initializer is used determines the type of the array being initialized. In an array creation expression, the array type immediately precedes the initializer, or is inferred from the expressions in the array initializer. In a field or variable declaration, the array type is the type of the field or variable being declared. When an array initializer is used in a field or variable declaration, such as

```
int[] a = {0, 2, 4, 6, 8};
```

it is simply shorthand for an equivalent array creation expression:

```
int[] a = new int[] {0, 2, 4, 6, 8};
```

For a single-dimensional array, the array initializer must consist of a sequence of expressions that are assignment compatible with the element type of the array. The expressions initialize array elements in increasing order, starting with the element at index 0. The number of expressions in the array initializer determines the length of the array instance being created. For example, the array initializer above creates an `int[]` instance of length 5 and then initializes the instance with the following values:

```
a[0] = 0; a[1] = 2; a[2] = 4; a[3] = 6; a[4] = 8;
```

For a multi-dimensional array, the array initializer must have as many levels of nesting as there are dimensions in the array. The outermost nesting level corresponds to the leftmost dimension and the innermost nesting level corresponds to the rightmost dimension. The length of each dimension of the array is determined by the number of elements at the corresponding nesting level in the array initializer. For each nested array initializer, the number of elements must be the same as the other array initializers at the same level. The example

```
int[,] b = {{0, 1}, {2, 3}, {4, 5}, {6, 7}, {8, 9}};
```

creates a two-dimensional array with a length of 5 for the leftmost dimension and a length of 2 for the rightmost dimension:

```
int[,] b = new int[5, 2];
```


www.it-ebooks.info

It then initializes the array instance with the following values:

```
b[0, 0] = 0; b[0, 1] = 1;
b[1, 0] = 2; b[1, 1] = 3;
b[2, 0] = 4; b[2, 1] = 5;
b[3, 0] = 6; b[3, 1] = 7;
b[4, 0] = 8; b[4, 1] = 9;
```

If a dimension other than the rightmost is given with length 0, the subsequent dimensions are assumed to also have length 0. The example

```
int[,] c = {};
```

creates a two-dimensional array with a length of 0 for both the leftmost and rightmost dimensions:

```
int[,] c = new int[0, 0];
```

When an array creation expression includes both explicit dimension lengths and an array initializer, the lengths must be constant expressions and the number of elements at each nesting level must match the corresponding dimension length. Here are some examples:

```
int i = 3;
int[] x = new int[3] {0, 1, 2};     // Okay
int[] y = new int[i] {0, 1, 2};     // Error: i not a constant
int[] z = new int[3] {0, 1, 2, 3};  // Error: length/initializer mismatch
```

Here, the initializer for `y` results in a compile-time error because the dimension length expression is not a constant, and the initializer for `z` results in a compile-time error because the length and the number of elements in the initializer do not agree.

www.it-ebooks.info

13. Interfaces

An interface defines a contract. A class or struct that implements an interface must adhere to its contract. An interface may inherit from multiple base interfaces, and a class or struct may implement multiple interfaces.

> **BILL WAGNER** Interfaces work best when they are small in scope and few in number. Larger interfaces create more work for implementers. Larger numbers of interfaces provide more opportunities for ambiguity and collisions.

> **ERIC LIPPERT** The trope that interfaces are contracts is undoubtedly both useful and frequently stated. It's worth pointing out an interface is actually a pretty weak way to represent a contract. All an interface tells you is which methods are available, what their names are, which types they take, and which types they return. Nothing whatsoever about the semantics of the operation is represented in the contract: that an object complying with this contract must be disposed aggressively, that `Drive()` throws an exception if not called after `StartEngine()`, that the first parameter must be `null` and the second parameter must be non-zero, and so on. All that stuff goes in the documentation, not somewhere that an analysis tool can dig into. The new code contract system that ships with version 4.0 of the CLR enables you both to specify contracts in more detail than interfaces do alone and to do interesting analysis on these contracts at compile time.

Interfaces can contain methods, properties, events, and indexers. The interface itself does not provide implementations for the members that it defines. The interface merely specifies the members that must be supplied by classes or structs that implement the interface.

13.1 Interface Declarations

An *interface-declaration* is a *type-declaration* (§9.6) that declares a new interface type.

www.it-ebooks.info

interface-declaration:
attributes$_{opt}$ *interface-modifiers*$_{opt}$ `partial`$_{opt}$ `interface`
identifier *variant-type-parameter-list*$_{opt}$ *interface-base*$_{opt}$
type-parameter-constraints-clauses$_{opt}$ *interface-body* `;`$_{opt}$

An *interface-declaration* consists of an optional set of *attributes* (§17), followed by an optional set of *interface-modifiers* (§13.1.1), followed by an optional `partial` modifier, followed by the keyword `interface` and an *identifier* that names the interface, followed by an optional *variant-type-parameter-list* specification (§13.1.3), followed by an optional *interface-base* specification (§13.1.4), followed by an optional *type-parameter-constraints-clauses* specification (§10.1.5), followed by an *interface-body* (§13.1.5), optionally followed by a semicolon.

13.1.1 Interface Modifiers

An *interface-declaration* may optionally include a sequence of interface modifiers:

interface-modifiers:
interface-modifier
interface-modifiers *interface-modifier*

interface-modifier:
`new`
`public`
`protected`
`internal`
`private`

It is a compile-time error for the same modifier to appear multiple times in an interface declaration.

The `new` modifier is permitted only on interfaces defined within a class. It specifies that the interface hides an inherited member by the same name, as described in §10.3.4.

The `public`, `protected`, `internal`, and `private` modifiers control the accessibility of the interface. Depending on the context in which the interface declaration occurs, only some of these modifiers may be permitted (§3.5.1).

13.1.2 partial Modifier

The `partial` modifier indicates that this *interface-declaration* is a partial type declaration. Multiple partial interface declarations with the same name within an enclosing namespace or type declaration combine to form one interface declaration, following the rules specified in §10.2.

www.it-ebooks.info

13.1.3 Variant Type Parameter Lists

Variant type parameter lists can occur only on interface and delegate types. The difference from ordinary *type-parameter-list*s is the optional *variance-annotation* on each type parameter.

variant-type-parameter-list:
    `<` *variant-type-parameters* `>`

variant-type-parameters:
    attributes$_{opt}$ *variance-annotation*$_{opt}$ *type-parameter*
    variant-type-parameters `,` *attributes*$_{opt}$ *variance-annotation*$_{opt}$ *type-parameter*

variance-annotation:
    `in`
    `out`

If the variance annotation is `out`, the type parameter is said to be ***covariant***. If the variance annotation is `in`, the type parameter is said to be ***contravariant***. If there is no variance annotation, the type parameter is said to be ***invariant***.

ERIC LIPPERT Covariance is the property that a mapping from a type argument to a generic type preserves assignment compatibility. For example, a string may be assigned to a variable of type `object`. A mapping from `T` to `IEnumerable<T>` preserves the assignment compatibility; an `IEnumerable<string>` can be assigned to a variable of `IEnumerable<object>`. Thus the nomenclature that "the type parameter is covariant" is a shorthand; the thing that is actually covariant is the relationship between the type argument and the constructed type. Properly we ought to say, "The mapping from —any reference type `T` to `IEnumerable<T>` is covariant in `T`" but that is rather a mouthful compared to simply saying, "`T` is covariant," and understanding that the longer statement is what is meant by it.

In the example

```
interface C<out X, in Y, Z>
{
    X M(Y y);

    Z P { get; set; }
}
```

`X` is covariant, `Y` is contravariant, and `Z` is invariant.

www.it-ebooks.info

13.1.3.1 *Variance Safety*

The occurrence of variance annotations in the type parameter list of a type restricts the places where types can occur within the type declaration.

A type `T` is ***output-unsafe*** if one of the following holds:

- `T` is a contravariant type parameter.
- `T` is an array type with an output-unsafe element type.
- `T` is an interface or delegate type $S<A_1, \ldots A_k>$ constructed from a generic type $S<X_1, \ldots X_k>$ where for at least one A_i one of the following holds:
  - X_i is covariant or invariant and A_i is output-unsafe.
  - X_i is contravariant or invariant and A_i is input-safe.

A type `T` is ***input-unsafe*** if one of the following holds:

- `T` is a covariant type parameter.
- `T` is an array type with an input-unsafe element type.
- `T` is an interface or delegate type $S<A_1, \ldots A_k>$ constructed from a generic type $S<X_1, \ldots X_k>$ where for at least one A_i one of the following holds:
  - X_i is covariant or invariant and A_i is input-unsafe.
  - X_i is contravariant or invariant and A_i is output-unsafe.

Intuitively, an output-unsafe type is prohibited in an output position, and an input-unsafe type is prohibited in an input position.

A type is ***output-safe*** if it is not output-unsafe, and ***input-safe*** if it is not input-unsafe.

13.1.3.2 *Variance Conversion*

The purpose of variance annotations is to provide for more lenient (but still type-safe) conversions to interface and delegate types. To this end, the definitions of implicit (§6.1) and explicit conversions (§6.2) make use of the notion of variance convertibility, which is defined as follows:

A type $T<A_1, \ldots, A_n>$ is variance convertible to a type $T<B_1, \ldots, B_n>$ if `T` is either an interface or a delegate type declared with the variant type parameters $T<X_1, \ldots, X_n>$, and for each variant type parameter X_i one of the following holds:

- X_i is covariant and an implicit reference or identity conversion exists from A_i to B_i.
- X_i is contravariant and an implicit reference or identity conversion exists from B_i to A_i.
- X_i is invariant and an identity conversion exists from A_i to B_i.

www.it-ebooks.info

JON SKEET I suspect many developers will never need to declare their own variant interfaces or delegates. In fact, my guess is that a lot of the time we won't even notice when we're *using* variance—it will just mean code that we would intuitively expect to work will now do so, even though it would have failed to compile in C# 3.0.

13.1.4 Base Interfaces

An interface can inherit from zero or more interface types, which are called the ***explicit base interfaces*** of the interface. When an interface has one or more explicit base interfaces, then in the declaration of that interface, the interface identifier is followed by a colon and a comma-separated list of base interface types.

interface-base:
: *interface-type-list*

For a constructed interface type, the explicit base interfaces are formed by taking the explicit base interface declarations on the generic type declaration, and substituting, for each *type-parameter* in the base interface declaration, the corresponding *type-argument* of the constructed type.

The explicit base interfaces of an interface must be at least as accessible as the interface itself (§3.5.4). For example, it is a compile-time error to specify a `private` or `internal` interface in the *interface-base* of a `public` interface.

It is a compile-time error for an interface to directly or indirectly inherit from itself.

VLADIMIR RESHETNIKOV For the purposes of this rule, type arguments (if any) are ignored. For instance, although `I<T>` and `I<I<T>>` are different types, the following declaration is still invalid:

```
interface I<T> : I<I<T>> { }
```

Conversely, it is perfectly valid for an interface to appear within a type argument for its base interface:

```
interface IA<T> { }
interface IB : IA<IB[]> { }     // Okay
```

The ***base interfaces*** of an interface are the explicit base interfaces and their base interfaces. In other words, the set of base interfaces is the complete transitive closure of the explicit

www.it-ebooks.info

base interfaces, their explicit base interfaces, and so on. An interface inherits all members of its base interfaces. In the example

```
interface IControl
{
    void Paint();
}

interface ITextBox : IControl
{
    void SetText(string text);
}

interface IListBox : IControl
{
    void SetItems(string[] items);
}

interface IComboBox : ITextBox, IListBox { }
```

the base interfaces of `IComboBox` are `IControl`, `ITextBox`, and `IListBox`.

In other words, the `IComboBox` interface above inherits members `SetText` and `SetItems` as well as `Paint`.

Every base interface of an interface must be output-safe (§13.1.3.1). A class or struct that implements an interface also implicitly implements all of the interface's base interfaces.

> **ERIC LIPPERT** "Inheritance" is an unfortunate choice of words for interfaces. One normally thinks of inheritance from a base as sharing implementation, which interfaces plainly do not do. I prefer to think of interfaces as *contracts that may specify other contracts that must also be fulfilled,* rather than as contracts that "inherit" other contracts.

> **JESSE LIBERTY** A naming convention has arisen of prefixing all interfaces with the letter "I" (`IControl`, `IWriteable`, `IClaudius`). There is no compelling reason to do so, yet it has persisted for at least a decade as a last remnant of "Hungarian" notation. Removing the "I", in the end, makes for more readable code.

13.1.5 Interface Body

The *interface-body* of an interface defines the members of the interface.

interface-body:
{ *interface-member-declarations*$_{opt}$ }

www.it-ebooks.info

13.2 Interface Members

The members of an interface are the members inherited from the base interfaces and the members declared by the interface itself.

interface-member-declarations:
interface-member-declaration
interface-member-declarations interface-member-declaration

interface-member-declaration:
interface-method-declaration
interface-property-declaration
interface-event-declaration
interface-indexer-declaration

An interface declaration may declare zero or more members. The members of an interface must be methods, properties, events, or indexers. An interface cannot contain constants, fields, operators, instance constructors, destructors, or types, nor can an interface contain static members of any kind.

All interface members implicitly have public access. It is a compile-time error for interface member declarations to include any modifiers. In particular, interfaces members cannot be declared with the modifiers `abstract`, `public`, `protected`, `internal`, `private`, `virtual`, `override`, or `static`.

JON SKEET It would occasionally be nice to be able to create an internal interface with internal members that could be implemented implicitly by internal methods. Currently an internal interface forces implementing types either to expose public methods or to use explicit interface implementation; in some cases, neither of these choices is particularly pleasant.

The example

```
public delegate void StringListEvent(IStringList sender);

public interface IStringList
{
    void Add(string s);

    int Count { get; }

    event StringListEvent Changed;

    string this[int index] { get; set; }
}
```

declares an interface that contains one each of the possible kinds of members: a method, a property, an event, and an indexer.

www.it-ebooks.info

An *interface-declaration* creates a new declaration space (§3.3), and the *interface-member-declarations* immediately contained by the *interface-declaration* introduce new members into this declaration space. The following rules apply to *interface-member-declaration*s:

- The name of a method must differ from the names of all properties and events declared in the same interface. In addition, the signature (§3.6) of a method must differ from the signatures of all other methods declared in the same interface, and two methods declared in the same interface may not have signatures that differ solely by `ref` and `out`.
- The name of a property or event must differ from the names of all other members declared in the same interface.
- The signature of an indexer must differ from the signatures of all other indexers declared in the same interface.

The inherited members of an interface are specifically not part of the declaration space of the interface. Thus an interface is allowed to declare a member with the same name or signature as an inherited member. When this occurs, the derived interface member is said to *hide* the base interface member. Hiding an inherited member is not considered an error, but it does cause the compiler to issue a warning. To suppress the warning, the declaration of the derived interface member must include a `new` modifier to indicate that the derived member is intended to hide the base member. This topic is discussed further in §3.7.1.2.

If a `new` modifier is included in a declaration that doesn't hide an inherited member, a warning is issued to that effect. This warning is suppressed by removing the `new` modifier.

Note that the members in class `object` are not, strictly speaking, members of any interface (§13.2). However, the members in class `object` are available via member lookup in any interface type (§7.4).

13.2.1 Interface Methods

Interface methods are declared using *interface-method-declaration*s:

interface-method-declaration:
attributes$_{opt}$ `new`$_{opt}$ *return-type* *identifier* *type-parameter-list*
`(` *formal-parameter-list*$_{opt}$ `)` *type-parameter-constraints-clauses*$_{opt}$ `;`

The *attributes*, *return-type*, *identifier*, and *formal-parameter-list* of an interface method declaration have the same meaning as those of a method declaration in a class (§10.6). An interface method declaration is not permitted to specify a method body, so the declaration always ends with a semicolon.

www.it-ebooks.info

VLADIMIR RESHETNIKOV In contrast to `class` or `struct` method declarations, the *identifier* of an *interface-method-declaration* can be the same as the name of the enclosing interface declaration. The same applies to *interface-property-declarations* and *interface-event-declarations.*

Each formal parameter type of an interface method must be input-safe (§13.1.3.1), and the return type must be either `void` or output-safe. Furthermore, each class type constraint, interface type constraint, and type parameter constraint on any type parameter of the method must be input-safe.

These rules ensure that any covariant or contravariant usage of the interface remains type-safe. For example,

```
interface I<out T> { void M<U>() where U : T; }
```

is illegal because the usage of `T` as a type parameter constraint on `U` is not input-safe.

Were this restriction not in place, it would be possible to violate type safety in the following manner:

```
class B {}
class D : B {}
class E : B {}
class C : I<D> { public void M<U>() {...} }
...
I<B> b = new C();
b.M<E>();
```

This is actually a call to `C.M<E>`. That call requires that `E` derive from `D`, however, so type safety would be violated here.

13.2.2 Interface Properties

Interface properties are declared using *interface-property-declaration*s:

interface-property-declaration:
attributes$_{opt}$ `new`$_{opt}$ *type* *identifier* `{` *interface-accessors* `}`

interface-accessors:
attributes$_{opt}$ `get` `;`
attributes$_{opt}$ `set` `;`
attributes$_{opt}$ `get` `;` *attributes*$_{opt}$ `set` `;`
attributes$_{opt}$ `set` `;` *attributes*$_{opt}$ `get` `;`

The *attributes*, *type*, and *identifier* of an interface property declaration have the same meaning as those of a property declaration in a class (§10.7).

www.it-ebooks.info

The accessors of an interface property declaration correspond to the accessors of a class property declaration (§10.7.2), except that the accessor body must always be a semicolon. Thus the accessors simply indicate whether the property is read-write, read-only, or write-only.

The type of an interface property must be output-safe if there is a `get` accessor, and must be input-safe if there is a `set` accessor.

13.2.3 Interface Events

Interface events are declared using *interface-event-declarations*:

interface-event-declaration:
attributes$_{opt}$ `new`$_{opt}$ `event` `type` *identifier* `;`

The *attributes*, *type*, and *identifier* of an interface event declaration have the same meaning as those of an event declaration in a class (§10.8).

The type of an interface event must be input-safe.

13.2.4 Interface Indexers

Interface indexers are declared using *interface-indexer-declaration*s:

interface-indexer-declaration:
attributes$_{opt}$ `new`$_{opt}$ *type* `this` `[` *formal-parameter-list* `]` `{` *interface-accessors* `}`

The *attributes*, *type*, and *formal-parameter-list* of an interface indexer declaration have the same meaning as those of an indexer declaration in a class (§10.9).

The accessors of an interface indexer declaration correspond to the accessors of a class indexer declaration (§10.9), except that the accessor body must always be a semicolon. Thus the accessors simply indicate whether the indexer is read-write, read-only, or write-only.

All the formal parameter types of an interface indexer must be input-safe. In addition, any `out` or `ref` formal parameter types must also be output-safe. Note that even `out` parameters are required to be input-safe, due to a limitation of the underlying execution platform.

The type of an interface indexer must be output-safe if there is a `get` accessor, and must be input-safe if there is a `set` accessor.

13.2.5 Interface Member Access

Interface members are accessed through member access (§7.6.4) and indexer access (§7.6.6.2) expressions of the form `I.M` and `I[A]`, where `I` is an interface type; `M` is a method, property, or event of that interface type; and `A` is an indexer argument list.

www.it-ebooks.info

For interfaces that are strictly single-inheritance (each interface in the inheritance chain has exactly zero or one direct base interface), the effects of the member lookup (§7.4), method invocation (§7.6.5.1), and indexer access (§7.6.6.2) rules are exactly the same as for classes and structs: More derived members hide less derived members with the same name or signature. However, for multiple-inheritance interfaces, ambiguities can occur when two or more unrelated base interfaces declare members with the same name or signature. This section shows several examples of such situations. In all cases, explicit casts can be used to resolve the ambiguities.

In the example

```
interface IList
{
    int Count { get; set; }
}

interface ICounter
{
    void Count(int i);
}

interface IListCounter : IList, ICounter { }

class C
{
    void Test(IListCounter x)
    {
        x.Count(1);                 // Error
        x.Count = 1;                // Error
        ((IList)x).Count = 1;       // Okay: invokes IList.Count.set
        ((ICounter)x).Count(1);     // Okay: invokes ICounter.Count
    }
}
```

the first two statements cause compile-time errors because the member lookup (§7.4) of `Count` in `IListCounter` is ambiguous. As illustrated by the example, the ambiguity is resolved by casting `x` to the appropriate base interface type. Such casts have no runtime costs—they merely consist of viewing the instance as a less derived type at compile time.

In the example

```
interface IInteger
{
    void Add(int i);
}

interface IDouble
{
    void Add(double d);
}

interface INumber : IInteger, IDouble { }
```


www.it-ebooks.info

```
class C
{
    void Test(INumber n)
    {
        n.Add(1);                   // Invokes IInteger.Add
        n.Add(1.0);                 // Only IDouble.Add is applicable
        ((IInteger)n).Add(1);       // Only IInteger.Add is a candidate
        ((IDouble)n).Add(1);        // Only IDouble.Add is a candidate
    }
}
```

the invocation `n.Add(1)` selects `IInteger.Add` by applying the overload resolution rules of §7.5.3. Similarly, the invocation `n.Add(1.0)` selects `IDouble.Add`. When explicit casts are inserted, there is only one candidate method and, therefore, no ambiguity.

In the example

```
interface IBase
{
    void F(int i);
}

interface ILeft: IBase
{
    new void F(int i);
}

interface IRight: IBase
{
    void G();
}

interface IDerived: ILeft, IRight {}

class A
{
    void Test(IDerived d) {
        d.F(1);                 // Invokes ILeft.F
        ((IBase)d).F(1);        // Invokes IBase.F
        ((ILeft)d).F(1);        // Invokes ILeft.F
        ((IRight)d).F(1);       // Invokes IBase.F
    }
}
```

the `IBase.F` member is hidden by the `ILeft.F` member. The invocation `d.F(1)` thus selects `ILeft.F`, even though `IBase.F` appears to not be hidden in the access path that leads through `IRight`.

The intuitive rule for hiding in multiple-inheritance interfaces is simply this: If a member is hidden in any access path, it is hidden in all access paths. Because the access path from `IDerived` to `ILeft` to `IBase` hides `IBase.F`, the member is also hidden in the access path from `IDerived` to `IRight` to `IBase`.

www.it-ebooks.info

13.3 Fully Qualified Interface Member Names

An interface member is sometimes referred to by its ***fully qualified name***. The fully qualified name of an interface member consists of the name of the interface in which the member is declared, followed by a dot, followed by the name of the member. The fully qualified name of a member references the interface in which the member is declared. For example, given the declarations

```
interface IControl
{
    void Paint();
}

interface ITextBox : IControl
{
    void SetText(string text);
}
```

the fully qualified name of `Paint` is `IControl.Paint` and the fully qualified name of `SetText` is `ITextBox.SetText`.

In the example above, it is not possible to refer to `Paint` as `ITextBox.Paint`.

When an interface is part of a namespace, the fully qualified name of an interface member includes the namespace name. For example:

```
namespace System
{
    public interface ICloneable
    {
        object Clone();
    }
}
```

Here, the fully qualified name of the `Clone` method is `System.ICloneable.Clone`.

13.4 Interface Implementations

Interfaces may be implemented by classes and structs. To indicate that a class or struct directly implements an interface, the interface identifier is included in the base class list of the class or struct. For example:

```
interface ICloneable
{
    object Clone();
}
```


www.it-ebooks.info

```
interface IComparable
{
    int CompareTo(object other);
}

class ListEntry : ICloneable, IComparable
{
    public object Clone() {...}

    public int CompareTo(object other) {...}
}
```

JON SKEET It bugs me that the interface implementation is usually so totally implicit. There's no indication that the `CompareTo` and `Clone` methods have anything to do with the interfaces for which they're providing implementations. When overriding virtual methods inherited from a base class, the `override` modifier makes this behavior obvious: Anyone refactoring a class and wanting to rename a method is informed that it's linked to a method elsewhere. No such indication is given for interfaces.

Unfortunately, the design of interface implementation wouldn't quite be consistent with adding a modifier to the declaration. The code declaring a method doesn't necessarily even know that it may be used to implement an interface:

```
interface IFoo { void Foo(); }
class Base
{
    public void Foo();
}
// IFoo.Foo is implemented by Base.Foo
class Derived : Base, IFoo { }
```

This is probably the most pragmatic approach to interfaces, but from a purity (or control freak) standpoint, it feels slightly wrong.

A class or struct that directly implements an interface also directly implements all of the interface's base interfaces implicitly. This is true even if the class or struct doesn't explicitly list all base interfaces in the base class list. For example:

```
interface IControl
{
    void Paint();
}

interface ITextBox : IControl
{
    void SetText(string text);
}
```


www.it-ebooks.info

```
class TextBox : ITextBox
{
    public void Paint() {...}

    public void SetText(string text) {...}
}
```

Here, class `TextBox` implements both `IControl` and `ITextBox`.

When a class `C` directly implements an interface, all classes derived from C also implement the interface implicitly. The base interfaces specified in a class declaration can be constructed interface types (§4.4). A base interface cannot be a type parameter on its own, although it can involve the type parameters that are in scope. The following code illustrates how a class can implement and extend constructed types:

```
class C<U, V> { }

interface I1<V> { }

class D : C<string, int>, I1<string> { }

class E<T> : C<int, T>, I1<T> { }
```

The base interfaces of a generic class declaration must satisfy the uniqueness rule described in §13.4.2.

13.4.1 Explicit Interface Member Implementations

For purposes of implementing interfaces, a class or struct may declare ***explicit interface member implementations***. An explicit interface member implementation is a method, property, event, or indexer declaration that references a fully qualified interface member name. For example:

```
interface IList<T>
{
      T[] GetElements();
}

interface IDictionary<K,V>
{
      V this[K key];

      void Add(K key, V value);
}

class List<T>: IList<T>, IDictionary<int,T>
{
      T[] IList<T>.GetElements() {...}

      T IDictionary<int,T>.this[int index] {...}

   void IDictionary<int,T>.Add(int index, T value) {...}
}
```


www.it-ebooks.info

Here `IDictionary<int,T>.this` and `IDictionary<int,T>.Add` are explicit interface member implementations.

In some cases, the name of an interface member may not be appropriate for the implementing class, in which case the interface member may be implemented using explicit interface member implementation. A class implementing a file abstraction, for example, would likely implement a `Close` member function that has the effect of releasing the file resource, and implement the `Dispose` method of the `IDisposable` interface using explicit interface member implementation:

```
interface IDisposable
{
    void Dispose();
}

class MyFile : IDisposable
{
    void IDisposable.Dispose()
    {
        Close();
    }

    public void Close()
    {
        // Do what's necessary to close the file
        System.GC.SuppressFinalize(this);
    }
}
```

It is not possible to access an explicit interface member implementation through its fully qualified name in a method invocation, property access, or indexer access. An explicit interface member implementation can be accessed only through an interface instance, and is in that case referenced simply by its member name.

It is a compile-time error for an explicit interface member implementation to include access modifiers, and it is a compile-time error to include the modifiers `abstract`, `virtual`, `override`, or `static`.

Explicit interface member implementations have different accessibility characteristics than other members. Because explicit interface member implementations are never accessible through their fully qualified names in a method invocation or a property access, they are, in a sense, private. However, because they can be accessed through an interface instance, they are, in a sense, also public.

VLADIMIR RESHETNIKOV For the purposes of accessibility constraints checking (see §3.5.4), explicit interface implementations are considered private.

www.it-ebooks.info

Explicit interface member implementations serve two primary purposes:

- Because explicit interface member implementations are not accessible through class or struct instances, they allow interface implementations to be excluded from the public interface of a class or struct. This is particularly useful when a class or struct implements an internal interface that is of no interest to a consumer of that class or struct.
- Explicit interface member implementations allow disambiguation of interface members with the same signature. Without explicit interface member implementations, it would be impossible for a class or struct to have different implementations of interface members with the same signature and return type, as would it be impossible for a class or struct to have any implementation at all of interface members with the same signature but with different return types.

JON SKEET The somewhat canonical example of this second case is `IEnumerable<T>`, which extends the contract of `IEnumerable`. That means implementations have to provide two methods:

```
IEnumerator GetEnumerator()
IEnumerator<T> GetEnumerator()
```

This is typically done by explicitly implementing the nongeneric `IEnumerable.GetEnumerator()` to call the generic version—which works because `IEnumerator<T>` and `IEnumerator` have the same kind of relationship.

For an explicit interface member implementation to be valid, the class or struct must name an interface in its base class list that contains a member whose fully qualified name, type, and parameter types exactly match those of the explicit interface member implementation. Thus, in the following class

```
class Shape: ICloneable
{
    object ICloneable.Clone() {...}

    int IComparable.CompareTo(object other) {...}  // Invalid
}
```

the declaration of `IComparable.CompareTo` results in a compile-time error because `IComparable` is not listed in the base class list of `Shape` and is not a base interface of `ICloneable`. Likewise, in the declarations

```
class Shape: ICloneable
{
    object ICloneable.Clone() {...}
}
```


www.it-ebooks.info

```
class Ellipse: Shape
{
    object ICloneable.Clone() {...}     // Invalid
}
```

the declaration of `ICloneable.Clone` in `Ellipse` results in a compile-time error because `ICloneable` is not explicitly listed in the base class list of `Ellipse`.

The fully qualified name of an interface member must reference the interface in which the member was declared. Thus, in the declarations

```
interface IControl
{
    void Paint();
}

interface ITextBox: IControl
{
    void SetText(string text);
}

class TextBox: ITextBox
{
    void IControl.Paint() {...}

    void ITextBox.SetText(string text) {...}
}
```

the explicit interface member implementation of `Paint` must be written as `IControl.Paint`.

13.4.2 Uniqueness of Implemented Interfaces

The interfaces implemented by a generic type declaration must remain unique for all possible constructed types. Without this rule, it would be impossible to determine the correct method to call for certain constructed types. For example, suppose a generic class declaration were permitted to be written as follows:

```
interface I<T>
{
    void F();
}

class X<U,V>: I<U>, I<V>    // Error: I<U> and I<V> conflict
{
    void I<U>.F() {...}
    void I<V>.F() {...}
}
```


www.it-ebooks.info

Were this permitted, it would be impossible to determine which code would execute in the following case:

```
I<int> x = new X<int,int>();
x.F();
```

To determine if the interface list of a generic type declaration is valid, the following steps are performed:

- Let `L` be the list of interfaces directly specified in a generic class, struct, or interface declaration `C`.
- Add to `L` any base interfaces of the interfaces already in `L`.
- Remove any duplicates from `L`.
- If any possible constructed type created from `C` would, after type arguments are substituted into `L`, cause two interfaces in `L` to be identical, then the declaration of `C` is invalid. Constraint declarations are not considered when determining all possible constructed types.

In the class declaration `X` above, the interface list `L` consists of `I<U>` and `I<V>`. The declaration is invalid because any constructed type with `U` and `V` being the same type would cause these two interfaces to be identical types.

It is possible for interfaces specified at different inheritance levels to unify:

```
interface I<T>
{
    void F();
}

class Base<U>: I<U>
{
    void I<U>.F() {...}
}

class Derived<U,V>: Base<U>, I<V>   // Okay
{
    void I<V>.F() {...}
}
```

This code is valid even though `Derived<U,V>` implements both `I<U>` and `I<V>`. The code

```
I<int> x = new Derived<int,int>();
x.F();
```

invokes the method in `Derived`, since `Derived<int,int>` effectively reimplements `I<int>` (§13.4.6).

www.it-ebooks.info

ERIC LIPPERT Although it is not legal to create classes such that two interfaces can become identical under construction, it is possible in C# 4.0 to create classes that are ambiguous in various, more subtle ways thanks to generic covariance and contravariance. Imagine, for example, a base class that implements `IEnumerable<object>` and a derived class that implements `IEnumerable<string>`. These were incompatible types in C# 3.0, but that is no longer the case. Because `IEnumerable<string>` is now convertible to `IEnumerable<object>`, it's not entirely clear which implementation will be called when a method of `IEnumerable<object>` is invoked on the derived class. These sorts of bizarre scenarios tend to expose implementation-defined behavior of the runtime and should be avoided whenever possible.

13.4.3 Implementation of Generic Methods

When a generic method implicitly implements an interface method, the constraints given for each method type parameter must be equivalent in both declarations (after any interface type parameters are replaced with the appropriate type arguments), where method type parameters are identified by ordinal positions, left to right.

When a generic method explicitly implements an interface method, however, no constraints are allowed on the implementing method. Instead, the constraints are inherited from the interface method:

```
interface I<A, B, C>
{
    void F<T>(T t) where T : A;
    void G<T>(T t) where T : B;
    void H<T>(T t) where T : C;
}

class C : I<object, C, string>
{
    public void F<T>(T t) {...}                        // Okay
    public void G<T>(T t) where T : C {...}            // Okay
    public void H<T>(T t) where T : string {...}       // Error
}
```

The method `C.F<T>` implicitly implements `I<object,C,string>.F<T>`. In this case, `C.F<T>` is not required (nor permitted) to specify the constraint `T: object` since `object` is an implicit constraint on all type parameters. The method `C.G<T>` implicitly implements `I<object,C,string>.G<T>` because the constraints match those in the interface, after the interface type parameters are replaced with the corresponding type arguments. The constraint for method `C.H<T>` is an error because sealed types (`string` in this case) cannot be used as constraints. Omitting the constraint would also be an error since constraints of

www.it-ebooks.info

implicit interface method implementations are required to match. Thus it is impossible to implicitly implement `I<object,C,string>.H<T>`. This interface method can be implemented only using an explicit interface member implementation:

```
class C: I<object,C,string>
{
    ...

    public void H<U>(U u) where U: class {...}

    void I<object,C,string>.H<T>(T t) {
        string s = t;     // Okay
        H<T>(t);
    }
}
```

In this example, the explicit interface member implementation invokes a public method having strictly weaker constraints. Note that the assignment from `t` to `s` is valid since `T` inherits a constraint of `T: string`, even though this constraint is not expressible in source code.

13.4.4 Interface Mapping

A class or struct must provide implementations of all members of the interfaces that are listed in the base class list of the class or struct. The process of locating implementations of interface members in an implementing class or struct is known as ***interface mapping***.

Interface mapping for a class or struct `C` locates an implementation for each member of each interface specified in the base class list of `C`. The implementation of a particular interface member `I.M`, where `I` is the interface in which the member `M` is declared, is determined by examining each class or struct `S`, starting with `C` and repeating for each successive base class of `C`, until a match is located:

- If `S` contains a declaration of an explicit interface member implementation that matches `I` and `M`, then this member is the implementation of `I.M`.
- Otherwise, if `S` contains a declaration of a nonstatic public member that matches `M`, then this member is the implementation of `I.M`. If more than one member matches, it is unspecified which member is the implementation of `I.M`. This situation can occur only if `S` is a constructed type where the two members as declared in the generic type have different signatures, but the type arguments make their signatures identical.

A compile-time error occurs if implementations cannot be located for all members of all interfaces specified in the base class list of `C`. Note that the members of an interface include those members that are inherited from base interfaces.

www.it-ebooks.info

For purposes of interface mapping, a class member `A` matches an interface member `B` when:

- `A` and `B` are methods, and the name, type, and formal parameter lists of `A` and `B` are identical.
- `A` and `B` are properties, the name and type of `A` and `B` are identical, and `A` has the same accessors as `B` (`A` is permitted to have additional accessors if it is not an explicit interface member implementation).
- `A` and `B` are events, and the name and type of `A` and `B` are identical.
- `A` and `B` are indexers, the type and formal parameter lists of `A` and `B` are identical, and `A` has the same accessors as `B` (`A` is permitted to have additional accessors if it is not an explicit interface member implementation).

Notable implications of the interface mapping algorithm are as follows:

- Explicit interface member implementations take precedence over other members in the same class or struct when determining the class or struct member that implements an interface member.
- Neither non-public nor static members participate in interface mapping.

In the example

```
interface ICloneable
{
    object Clone();
}

class C : ICloneable
{
    object ICloneable.Clone() {...}

    public object Clone() {...}
}
```

the `ICloneable.Clone` member of `C` becomes the implementation of `Clone` in `ICloneable` because explicit interface member implementations take precedence over other members.

If a class or struct implements two or more interfaces containing a member with the same name, type, and parameter types, it is possible to map each of those interface members onto a single class or struct member. For example:

```
interface IControl
{
    void Paint();
}
```


www.it-ebooks.info

```
interface IForm
{
    void Paint();
}

class Page : IControl, IForm
{
    public void Paint() {...}
}
```

Here, the `Paint` methods of both `IControl` and `IForm` are mapped onto the `Paint` method in `Page`. It is, of course, also possible to have separate explicit interface member implementations for the two methods.

If a class or struct implements an interface that contains hidden members, then some members must necessarily be implemented through explicit interface member implementations. For example:

```
interface IBase
{
    int P { get; }
}

interface IDerived : IBase
{
    new int P();
}
```

An implementation of this interface would require at least one explicit interface member implementation, and would take one of the following forms:

```
class C : IDerived
{
    int IBase.P { get {...} }

    int IDerived.P() {...}
}

class C : IDerived
{
    public int P { get {...} }

    int IDerived.P() {...}
}

class C : IDerived
{
    int IBase.P { get {...} }

    public int P() {...}
}
```


www.it-ebooks.info

When a class implements multiple interfaces that have the same base interface, there can be only one implementation of the base interface. In the example

```
interface IControl
{
    void Paint();
}

interface ITextBox : IControl
{
    void SetText(string text);
}

interface IListBox : IControl
{
    void SetItems(string[] items);
}

class ComboBox : IControl, ITextBox, IListBox
{
    void IControl.Paint() {...}

    void ITextBox.SetText(string text) {...}

    void IListBox.SetItems(string[] items) {...}
}
```

it is not possible to have separate implementations for the `IControl` named in the base class list, the `IControl` inherited by `ITextBox`, and the `IControl` inherited by `IListBox`. Indeed, there is no notion of a separate identity for these interfaces. Rather, the implementations of `ITextBox` and `IListBox` share the same implementation of `IControl`, and `ComboBox` is simply considered to implement three interfaces: `IControl`, `ITextBox`, and `IListBox`.

The members of a base class participate in interface mapping. In the example

```
interface Interface1
{
    void F();
}

class Class1
{
    public void F() { }

    public void G() { }
}

class Class2 : Class1, Interface1
{
    new public void G() { }
}
```

the method `F` in `Class1` is used in `Class2`'s implementation of `Interface1`.

www.it-ebooks.info

13.4.5 Interface Implementation Inheritance

A class inherits all interface implementations provided by its base classes.

> **BILL WAGNER** Eric's earlier comment about "inheritance" being an unfortunate word choice is even more true in this section. Interface methods behave differently than either virtual or non-virtual methods declared in base classes. It can take some time to get your mind around exactly which method is the best choice when multiple classes in a hierarchy declare implementation of an interface. As you read this section, you'll see many different rules for selecting the best method when that method is defined in an interface.

Without explicitly ***reimplementing*** an interface, a derived class cannot in any way alter the interface mappings it inherits from its base classes. For example, in the declarations

```
interface IControl
{
    void Paint();
}

class Control : IControl
{
    public void Paint() {...}
}

class TextBox : Control
{
    new public void Paint() {...}
}
```

the `Paint` method in `TextBox` hides the `Paint` method in `Control`, but it does not alter the mapping of `Control.Paint` onto `IControl.Paint`, and calls to `Paint` through class instances and interface instances will have the following effects:

```
Control c = new Control();
TextBox t = new TextBox();
IControl ic = c;
IControl it = t;
c.Paint();              // Invokes Control.Paint();
t.Paint();              // Invokes TextBox.Paint();
ic.Paint();             // Invokes Control.Paint();
it.Paint();             // Invokes Control.Paint();
```


www.it-ebooks.info

However, when an interface method is mapped onto a virtual method in a class, it is possible for derived classes to override the virtual method and alter the implementation of the interface. For example, after rewriting the declarations above to

```
interface IControl
{
    void Paint();
}

class Control: IControl
{
    public virtual void Paint() {...}
}

class TextBox: Control
{
    public override void Paint() {...}
}
```

the following effects will now be observed:

```
Control c = new Control();
TextBox t = new TextBox();
IControl ic = c;
IControl it = t;
c.Paint();              // Invokes Control.Paint();
t.Paint();              // Invokes TextBox.Paint();
ic.Paint();             // Invokes Control.Paint();
it.Paint();             // Invokes TextBox.Paint();
```

Since explicit interface member implementations cannot be declared as `virtual`, it is not possible to override an explicit interface member implementation. However, it is perfectly valid for an explicit interface member implementation to call another method, and that other method can be declared as `virtual` to allow derived classes to override it. For example:

```
interface IControl
{
    void Paint();
}

class Control: IControl
{
    void IControl.Paint() { PaintControl(); }

    protected virtual void PaintControl() {...}
}

class TextBox: Control
{
    protected override void PaintControl() {...}
}
```


www.it-ebooks.info

Here, classes derived from `Control` can specialize the implementation of `IControl.Paint` by overriding the `PaintControl` method.

13.4.6 Interface Reimplementation

A class that inherits an interface implementation is permitted to ***reimplement*** the interface by including it in the base class list.

A reimplementation of an interface follows exactly the same interface mapping rules as an initial implementation of an interface. Thus the inherited interface mapping has no effect whatsoever on the interface mapping established for the reimplementation of the interface. For example, in the declarations

```
interface IControl
{
    void Paint();
}

class Control: IControl
{
    void IControl.Paint() {...}
}

class MyControl: Control, IControl
{
    public void Paint() {}
}
```

the fact that `Control` maps `IControl.Paint` onto `Control.IControl.Paint` doesn't affect the reimplementation in `MyControl`, which maps `IControl.Paint` onto `MyControl.Paint`.

Inherited public member declarations and inherited explicit interface member declarations participate in the interface mapping process for reimplemented interfaces. For example:

```
interface IMethods
{
    void F();
    void G();
    void H();
    void I();
}

class Base: IMethods
{
    void IMethods.F() {}
    void IMethods.G() {}
    public void H() {}
    public void I() {}
}
```


www.it-ebooks.info

```
class Derived: Base, IMethods
{
    public void F() {}

    void IMethods.H() {}
}
```

Here, the implementation of `IMethods` in `Derived` maps the interface methods onto `Derived.F`, `Base.IMethods.G`, `Derived.IMethods.H`, and `Base.I`.

When a class implements an interface, it implicitly also implements all of that interface's base interfaces. Likewise, a reimplementation of an interface is also implicitly a reimplementation of all of the interface's base interfaces. For example:

```
interface IBase
{
    void F();
}

interface IDerived: IBase
{
    void G();
}

class C: IDerived
{
    void IBase.F() {...}

    void IDerived.G() {...}
}

class D: C, IDerived
{
    public void F() {...}

    public void G() {...}
}
```

Here, the reimplementation of `IDerived` also reimplements `IBase`, mapping `IBase.F` onto `D.F`.

JON SKEET While this sort of thing is genuinely useful on certain occasions, it should typically be avoided because it is a source of great confusion. Usually, every call using the same method name and the same argument list on the same object should result in the same method being invoked. There are any number of ways to stray from this happy situation, and they should all be used with extreme caution, and only where absolutely necessary.

www.it-ebooks.info

13.4.7 Abstract Classes and Interfaces

Like a nonabstract class, an abstract class must provide implementations of all members of the interfaces that are listed in the base class list of the class. However, an abstract class is permitted to map interface methods onto abstract methods. For example:

```
interface IMethods
{
    void F();
    void G();
}

abstract class C: IMethods
{
    public abstract void F();
    public abstract void G();
}
```

Here, the implementation of `IMethods` maps `F` and `G` onto abstract methods, which must be overridden in nonabstract classes that derive from `C`.

Note that explicit interface member implementations cannot be abstract, but explicit interface member implementations are, of course, permitted to call abstract methods. For example:

```
interface IMethods
{
    void F();
    void G();
}

abstract class C: IMethods
{
    void IMethods.F() { FF(); }

    void IMethods.G() { GG(); }

    protected abstract void FF();

    protected abstract void GG();
}
```

Here, nonabstract classes that derive from `C` would be required to override `FF` and `GG`, thus providing the actual implementation of `IMethods`.

www.it-ebooks.info

This page intentionally left blank

www.it-ebooks.info

14. Enums

An ***enum type*** is a distinct value type (§4.1) that declares a set of named constants.

> **JON SKEET** Notably, these "named constants" are effectively just numbers. They cannot express behavior, contrary to almost everything else in C#. This is one of the *very* few areas where Java is more expressive than C#. In Java, enums have a lot more power: An enum can declare methods and then override them for specific values, for example. While it's possible to emulate some of the features of Java enums within C#, language (and framework) support in a future version would be extremely welcome.

The example

```
enum Color
{
    Red,
    Green,
    Blue
}
```

declares an enum type named `Color` with members `Red`, `Green`, and `Blue`.

14.1 Enum Declarations

An enum declaration declares a new enum type. An enum declaration begins with the keyword `enum`, and defines the name, accessibility, underlying type, and members of the enum.

enum-declaration:
attributes$_{opt}$ *enum-modifiers*$_{opt}$ `enum` *identifier* *enum-base*$_{opt}$ *enum-body* `;`$_{opt}$

enum-base:
`:` *integral-type*

enum-body:
`{` *enum-member-declarations*$_{opt}$ `}`
`{` *enum-member-declarations* `,` `}`

www.it-ebooks.info

Each enum type has a corresponding integral type called the ***underlying type*** of the enum type. This underlying type must be able to represent all the enumerator values defined in the enumeration. An enum declaration may explicitly declare an underlying type of `byte`, `sbyte`, `short`, `ushort`, `int`, `uint`, `long`, or `ulong`. Note that `char` cannot be used as an underlying type. An enum declaration that does not explicitly declare an underlying type has an underlying type of `int`.

> **JON SKEET** This is one of the only places in the language where you cannot replace the "shorthand" version of a type with its full equivalent. For example, you cannot declare the `Color` enum as follows:
>
> ```
> enum Color: System.Int64 { ... }
> ```


The example

```
enum Color : long
{
    Red,
    Green,
    Blue
}
```

declares an enum with an underlying type of `long`. A developer might choose to use an underlying type of `long`, as in the example, to enable the use of values that are in the range of `long` but not in the range of `int`, or to preserve this option for the future.

14.2 Enum Modifiers

An *enum-declaration* may optionally include a sequence of enum modifiers:

enum-modifiers:
    enum-modifier
    enum-modifiers enum-modifier

enum-modifier:
    `new`
    `public`
    `protected`
    `internal`
    `private`

It is a compile-time error for the same modifier to appear multiple times in an enum declaration.

www.it-ebooks.info

The modifiers of an enum declaration have the same meaning as those of a class declaration (§10.1.1). Note, however, that the `abstract` and `sealed` modifiers are not permitted in an enum declaration. Enums cannot be abstract and do not permit derivation.

14.3 Enum Members

The body of an enum type declaration defines zero or more enum members, which are the named constants of the enum type. No two enum members can have the same name.

enum-member-declarations:
    enum-member-declaration
    enum-member-declarations `,` *enum-member-declaration*

enum-member-declaration:
    attributes$_{opt}$ *identifier*
    attributes$_{opt}$ *identifier* `=` *constant-expression*

> **VLADIMIR RESHETNIKOV** In the Microsoft implementation of C#, an enum member cannot have the name `value__`, because this name is reserved for the internal representation of enums.

Each enum member has an associated constant value. The type of this value is the underlying type for the containing enum. The constant value for each enum member must be in the range of the underlying type for the enum. The example

```
enum Color : uint
{
    Red = -1,
    Green = -2,
    Blue = -3
}
```

results in a compile-time error because the constant values `-1`, `-2`, and `-3` are not in the range of the underlying integral type `uint`.

Multiple enum members may share the same associated value. The example

```
enum Color
{
    Red,
    Green,
    Blue,

    Max = Blue
}
```


www.it-ebooks.info

shows an enum in which two enum members—`Blue` and `Max`—have the same associated value.

The associated value of an enum member is assigned either implicitly or explicitly. If the declaration of the enum member has a *constant-expression* initializer, the value of that constant expression, implicitly converted to the underlying type of the enum, is the associated value of the enum member. If the declaration of the enum member has no initializer, its associated value is set implicitly, as follows:

- If the enum member is the first enum member declared in the enum type, its associated value is `0`.
- Otherwise, the associated value of the enum member is obtained by increasing the associated value of the textually preceding enum member by 1. This increased value must be within the range of values that can be represented by the underlying type; otherwise, a compile-time error occurs.

JON SKEET Using the default values is almost always the wrong thing to do for a `[Flags]` enum where typically values should be `1`, `2`, `4`, `8`, and so on—often with a "None" value for `0`. The language could have helped to avoid developers from accidentally using inappropriate values, but it's hard to judge how well this goal could be achieved without adding more complexity than is merited.

BILL WAGNER I would prefer to see the following example implemented using extension methods. "`Color.Red.StringFromColor()`" just reads better to me than "`StringFromColor(Color.Red)`".

That technique also starts to address Jon's comment about not being able to express behavior.

The example

```
using System;

enum Color
{
    Red,
    Green = 10,
    Blue
}
```


www.it-ebooks.info

```
class Test
{
    static void Main()
    {
        Console.WriteLine(StringFromColor(Color.Red));
        Console.WriteLine(StringFromColor(Color.Green));
        Console.WriteLine(StringFromColor(Color.Blue));
    }

    static string StringFromColor(Color c)
    {
        switch (c)
        {
            case Color.Red:
                return String.Format("Red = {0}", (int)c);

            case Color.Green:
                return String.Format("Green = {0}", (int)c);

            case Color.Blue:
                return String.Format("Blue = {0}", (int)c);

            default:
                return "Invalid color";
        }
    }
}
```


prints out the enum member names and their associated values. The output is

```
Red = 0
Green = 10
Blue = 11
```

for the following reasons:

- The enum member `Red` is automatically assigned the value zero (since it has no initializer and is the first enum member).
- The enum member `Green` is explicitly given the value `10`.
- The enum member `Blue` is automatically assigned the value 1 greater than the member that textually precedes it.

The associated value of an enum member may not, directly or indirectly, use the value of its own associated enum member. Other than this circularity restriction, enum member initializers may freely refer to other enum member initializers, regardless of their textual position. Within an enum member initializer, values of other enum members are always treated as having the type of their underlying type, so that casts are not necessary when referring to other enum members.

The example

www.it-ebooks.info

```
enum Circular
{
    A = B,
    B
}
```

results in a compile-time error because the declarations of `A` and `B` are circular. `A` depends on `B` explicitly, and `B` depends on `A` implicitly.

Enum members are named and scoped in a manner exactly analogous to fields within classes. The scope of an enum member is the body of its containing enum type. Within that scope, enum members can be referred to by their simple names. From all other code, the name of an enum member must be qualified with the name of its enum type. Enum members do not have any declared accessibility—an enum member is accessible if its containing enum type is accessible.

14.4 The System.Enum Type

The type `System.Enum` is the abstract base class of all enum types (this is distinct and different from the underlying type of the enum type), and the members inherited from `System.Enum` are available in any enum type. A boxing conversion (§4.3.1) exists from any enum type to `System.Enum`, and an unboxing conversion (§4.3.2) exists from `System.Enum` to any enum type.

Note that `System.Enum` is not itself an *enum-type*. Rather, it is a *class-type* from which all *enum-type*s are derived. The type `System.Enum` inherits from the type `System.ValueType` (§4.1.1), which in turn inherits from type `object`. At runtime, a value of type `System.Enum` can be `null` or a reference to a boxed value of any enum type.

14.5 Enum Values and Operations

Each enum type defines a distinct type; an explicit enumeration conversion (§6.2.2) is required to convert between an enum type and an integral type, or between two enum types. The set of values that an enum type can take on is not limited by its enum members. In particular, any value of the underlying type of an enum can be cast to the enum type, and is a distinct valid value of that enum type.

Enum members have the type of their containing enum type (except within other enum member initializers; see §14.3). The value of an enum member declared in enum type `E` with associated value `v` is `(E)v`.

www.it-ebooks.info

The following operators can be used on values of enum types: `==`, `!=`, `<`, `>`, `<=`, `>=` (§7.10.5), binary `+` (§7.8.4), binary `-` (§7.8.5), `^`, `&`, `|` (§7.11.2), `~` (§7.7.4), and `++` and `--` (§7.6.9 and §7.7.5).

> **JON SKEET** Some of these operators really make sense just for enums decorated with `[Flags]`, but the language does not enforce this limitation in any sense.

Every enum type automatically derives from the class `System.Enum` (which, in turn, derives from `System.ValueType` and `object`). Thus inherited methods and properties of this class can be used on values of an enum type.

www.it-ebooks.info

This page intentionally left blank

www.it-ebooks.info

15. Delegates

Delegates enable scenarios that other languages—such as C++, Pascal, and Modula—have addressed with function pointers. Unlike C++ function pointers, however, delegates are fully object oriented. Also, unlike C++ pointers to member functions, delegates encapsulate both an object instance and a method.

> **BILL WAGNER** I experience a strange sense of amazement upon reading this small chapter in the language specification. Delegates have been part of C# since the 1.0 release. At that time, most of the C# community (including me) saw delegates as a little extra ceremony around events. Ten years later, I can't imagine the C# language without delegates. They are something most of us use every day in LINQ queries, function composition, closures, and more. Delegates and the concept of treating code as data are an integral part of so much of the modern .NET ecosystem that I can't imagine programming in a language that does not allow me to express the concept of functions (and actions) as data.

A delegate declaration defines a class that is derived from the class `System.Delegate`. A delegate instance encapsulates an invocation list, which is a list of one or more methods, each of which is referred to as a callable entity. For instance methods, a callable entity consists of an instance and a method on that instance. For static methods, a callable entity consists of just a method. Invoking a delegate instance with an appropriate set of arguments causes each of the delegate's callable entities to be invoked with the given set of arguments.

An interesting and useful property of a delegate instance is that it does not know or care about the classes of the methods it encapsulates; all that matters is that those methods be compatible (§15.1) with the delegate's type. This makes delegates perfectly suited for "anonymous" invocation.

www.it-ebooks.info

ERIC LIPPERT Delegates are typically quite confusing the first time a developer encounters them. I like to think of a delegate type as something akin to an interface with exactly one method on it; an instance of a delegate type is an object that implements this method.

15.1 Delegate Declarations

A *delegate-declaration* is a *type-declaration* (§9.6) that declares a new delegate type.

delegate-declaration:
attributes$_{opt}$ *delegate-modifiers*$_{opt}$ `delegate` *return-type*
identifier *variant-type-parameter-list*$_{opt}$
`(` *formal-parameter-list*$_{opt}$ `)` *type-parameter-constraints-clauses*$_{opt}$ `;`

delegate-modifiers:
delegate-modifier
delegate-modifiers *delegate-modifier*

delegate-modifier:
`new`
`public`
`protected`
`internal`
`private`

It is a compile-time error for the same modifier to appear multiple times in a delegate declaration.

The `new` modifier is permitted only on delegates declared within another type, in which case it specifies that such a delegate hides an inherited member by the same name, as described in §10.3.4.

The `public`, `protected`, `internal`, and `private` modifiers control the accessibility of the delegate type. Depending on the context in which the delegate declaration occurs, some of these modifiers may not be permitted (§3.5.1).

The delegate's type name is *identifier*.

The optional *formal-parameter-list* specifies the parameters of the delegate, and the *return-type* indicates the return type of the delegate.

www.it-ebooks.info

The optional *variant-type-parameter-list* (§13.1.3) specifies the type parameters to the delegate itself.

The return type of a delegate type must be either `void` or output-safe (§13.1.3.1).

All of the formal parameter types of a delegate type must be input-safe. Additionally, any `out` or `ref` parameter types must be output-safe. Note that even `out` parameters are required to be input-safe, due to a limitation of the underlying execution platform.

> **ERIC LIPPERT** The limitation mentioned here is that `out` parameters are actually implemented as `ref` parameters behind the scenes: You can both read and write `out` parameters. The rule enforced by the compiler is that an `out` parameter must be written to before it is read from, but that is a rule of the C# language, not of the runtime. If `out` parameters were truly "write-only" and this restriction was enforced by the runtime, then they could, in theory, be made covariant. Keep that point in mind the next time you design a new type system.

Delegate types in C# are name equivalent, not structurally equivalent. Specifically, two different delegate types that have the same parameter lists and return type are considered different delegate types. However, instances of two distinct but structurally equivalent delegate types may compare as equal (§7.9.8).

For example:

```
delegate int D1(int i, double d);

class A
{
    public static int M1(int a, double b) {...}
}

class B
{
    delegate int D2(int c, double d);

    public static int M1(int f, double g) {...}

    public static void M2(int k, double l) {...}

    public static int M3(int g) {...}

    public static void M4(int g) {...}
}
```

The delegate types `D1` and `D2` are both compatible with the methods `A.M1` and `B.M1`, since they have the same return type and parameter list; however, these delegate types are two different types, so they are not interchangeable. The delegate types `D1` and `D2` are

www.it-ebooks.info

incompatible with the methods `B.M2`, `B.M3`, and `B.M4`, since they have different return types or parameter lists.

> **ERIC LIPPERT** When you're designing a new type system, you don't know exactly how it will be used in the future. One might imagine future scenarios involving different categories of delegates (such as "delegate to method with observable side effects" or "delegate to method that can be called concurrently"). It would then make sense to disallow assignments between these categories. In practice, as it turned out, structural typing on delegates is very useful; there is really no semantic difference between `Predicate<int>` and `Func<int, bool>` that needs to be enforced by the runtime. If we were to do it all over again, delegate types would be more structurally compatible than they are today.

Like other generic type declarations, type arguments must be given to create a constructed delegate type. The parameter types and return type of a constructed delegate type are created by substituting, for each type parameter in the delegate declaration, the corresponding type argument of the constructed delegate type. The resulting return type and parameter types are used in determining which methods are compatible with a constructed delegate type. For example:

```
delegate bool Predicate<T>(T value);

class X
{
    static bool F(int i) {...}

    static bool G(string s) {...}
}
```

The delegate type `Predicate<int>` is compatible with the method `X.F` and the delegate type `Predicate<string>` is compatible with the method `X.G`.

The only way to declare a delegate type is via a *delegate-declaration*. A delegate type is a class type that is derived from `System.Delegate`. Delegate types are implicitly `sealed`, so it is not permissible to derive any type from a delegate type. It is also not permissible to derive a non-delegate class type from `System.Delegate`. Note that `System.Delegate` is not itself a delegate type; it is a class type from which all delegate types are derived.

C# provides special syntax for delegate instantiation and invocation. Except for instantiation, any operation that can be applied to a class or class instance can also be applied to a delegate class or instance, respectively. In particular, it is possible to access members of the `System.Delegate` type via the usual member access syntax.

www.it-ebooks.info

The set of methods encapsulated by a delegate instance is called an invocation list. When a delegate instance is created (§15.2) from a single method, it encapsulates that method, and its invocation list contains only one entry. However, when two non-null delegate instances are combined, their invocation lists are concatenated—in the order left operand then right operand—to form a new invocation list, which contains two or more entries.

> **JON SKEET** This single-cast/multicast duality is terribly awkward in some ways—and wonderfully useful in others. For example, it makes sense to talk about "the target" of a single-cast delegate, whereas each entry in the invocation list for a multicast delegate may have a separate target. This rarely presents a practical problem, but can make hard to describe delegates both simply and accurately.

Delegates are combined using the binary + (§7.8.4) and += operators (§7.17.2). A delegate can be removed from a combination of delegates by using the binary - (§7.8.5) and -= operators (§7.17.2). Delegates can also be compared for equality (§7.10.8).

The following example shows the instantiation of a number of delegates, and their corresponding invocation lists:

```
delegate void D(int x);

class C
{
    public static void M1(int i) {...}
    public static void M2(int i) {...}

}

class Test
{
    static void Main()
    {
        D cd1 = new D(C.M1);        // M1
        D cd2 = new D(C.M2);        // M2
        D cd3 = cd1 + cd2;          // M1 + M2
        D cd4 = cd3 + cd1;          // M1 + M2 + M1
        D cd5 = cd4 + cd3;          // M1 + M2 + M1 + M1 + M2
    }

}
```

When `cd1` and `cd2` are instantiated, they each encapsulate one method. When `cd3` is instantiated, it has an invocation list of two methods, `M1` and `M2`, in that order. `cd4`'s invocation list contains `M1`, `M2`, and `M1`, in that order. Finally, `cd5`'s invocation list contains `M1`, `M2`, `M1`, `M1`, and `M2`, in that order. For more examples of combining (as well as removing) delegates, see §15.4.

www.it-ebooks.info

15.2 Delegate Compatibility

A method or delegate `M` is ***compatible*** with a delegate type `D` if all of the following are true:

- `D` and `M` have the same number of parameters, and each parameter in `D` has the same `ref` or `out` modifiers as the corresponding parameter in `M`.
- For each value parameter (a parameter with no `ref` or `out` modifier), an identity conversion (§6.1.1) or implicit reference conversion (§6.1.6) exists from the parameter type in `D` to the corresponding parameter type in `M`.
- For each `ref` or `out` parameter, the parameter type in `D` is the same as the parameter type in `M`.
- An identity or implicit reference conversion exists from the return type of `M` to the return type of `D`.

15.3 Delegate Instantiation

An instance of a delegate is created by a *delegate-creation-expression* (§7.6.10.5) or a conversion to a delegate type. The newly created delegate instance then refers to either

- The static method referenced in the *delegate-creation-expression*, or
- The target object (which cannot be `null`) and instance method referenced in the *delegate-creation-expression*, or
- Another delegate.

For example:

```
delegate void D(int x);

class C
{
    public static void M1(int i) {...}
    public void M2(int i) {...}
}

class Test
{
    static void Main()
    {
        D cd1 = new D(C.M1);        // Static method
        C t = new C();
        D cd2 = new D(t.M2);        // Instance method
        D cd3 = new D(cd2);         // Another delegate
    }
}
```


www.it-ebooks.info

Once instantiated, delegate instances always refer to the same target object and method. Remember, when two delegates are combined, or one is removed from another, a new delegate results with its own invocation list; the invocation lists of the delegates combined or removed remain unchanged.

15.4 Delegate Invocation

C# provides special syntax for invoking a delegate. When a non-null delegate instance whose invocation list contains one entry is invoked, it invokes the one method with the same arguments it was given, and returns the same value as the referred to method. (See §7.6.5.3 for detailed information on delegate invocation.) If an exception occurs during the invocation of such a delegate, and that exception is not caught within the method that was invoked, the search for an exception catch clause continues in the method that called the delegate, as if that method had directly called the method to which that delegate referred.

Invocation of a delegate instance whose invocation list contains multiple entries proceeds by invoking each of the methods in the invocation list, synchronously, in order. Each method so called is passed the same set of arguments as was given to the delegate instance. If such a delegate invocation includes reference parameters (§10.6.1.2), each method invocation will occur with a reference to the same variable; changes to that variable by one method in the invocation list will be visible to methods further down the invocation list. If the delegate invocation includes output parameters or a return value, their final value will come from the invocation of the last delegate in the list.

If an exception occurs during processing of the invocation of such a delegate, and that exception is not caught within the method that was invoked, the search for an exception catch clause continues in the method that called the delegate, and any methods further down the invocation list are not invoked.

BILL WAGNER This behavior is why, in the general case, you should strive to create methods that implement delegates that do not throw exceptions under any circumstances. It introduces errors from which you likely cannot safely recover. This consideration is less important when you know your delegate will be used in a single-cast scenario only.

CHRISTIAN NAGEL There's a way to deal with exceptions that are thrown from handler methods that are referenced by the delegate. Instead of invoking the delegate instance directly, the `GetInvocationList` method of the delegate can be used to invoke each delegate of the invocation list separately. This invocation can be guarded from a `try-catch` block. In case of an exception, one way to deal with the failing handler method is to remove it from the invocation list.

www.it-ebooks.info

Attempting to invoke a delegate instance whose value is `null` results in an exception of type `System.NullReferenceException`.

JON SKEET In some ways, this outcome makes perfect sense; in other ways, it's inconsistent with a null reference's status as the normal representation of an empty invocation list. This confusion could perhaps have been avoided by each delegate type having a *static* `Invoke` method—perhaps even an extension method—that could have performed the appropriate nullity check. This solution would be more elegant than the C# compiler automatically performing a null check on each invocation expression. Admittedly, things get more complicated when the delegate type has a non-void return type or uses an `out` parameter.

Whatever the best solution would have been, it's undeniably a pain in the neck to have to check for nullity everywhere that you want to invoke a delegate.

ERIC LIPPERT A common pattern for "thread-safe" delegates is to do something like this:

```
var temp = this.mydelegate;
if (temp != null) temp();
```

instead of the more obvious solution:

```
if (this.mydelegate != null) this.mydelegate();
```

The former is safer because if `mydelegate` can be changed on another thread, then it might be changed to `null` between the test and the invocation. Moreover, this approach eliminates only one race. Suppose we are using the former code. The temporary caches the current value of `mydelegate`. On that other thread, `mydelegate` is set to `null`, and the global state that the previous contents of `mydelegate` needs to execute successfully is destroyed. But it is the previous contents that are about to be invoked! If your delegates (particularly delegates associated with events) are susceptible to this problem, then some other mechanism must be implemented to ensure that nothing bad happens if this race occurs. Probably the best thing to do is to write the code to ensure that if a "stale" delegate is invoked, nothing bad happens.

The following example shows how to instantiate, combine, remove, and invoke delegates:

```
using System;

delegate void D(int x);
```


www.it-ebooks.info

```
class C
{
    public static void M1(int i)
    {
        Console.WriteLine("C.M1: " + i);
    }

    public static void M2(int i)
    {
        Console.WriteLine("C.M2: " + i);
    }

    public void M3(int i)
    {
        Console.WriteLine("C.M3: " + i);
    }
}

class Test
{
    static void Main()
    {
         D cd1 = new D(C.M1);
        cd1(-1);              // Call M1

        D cd2 = new D(C.M2);
        cd2(-2);              // Call M2

        D cd3 = cd1 + cd2;
        cd3(10);              // Call M1 then M2

        cd3 += cd1;
        cd3(20);              // Call M1, M2, then M1

        C c = new C();
        D cd4 = new D(c.M3);
        cd3 += cd4;
        cd3(30);              // Call M1, M2, M1, then M3

        cd3 -= cd1;           // Remove last M1
        cd3(40);              // Call M1, M2, then M3

        cd3 -= cd4;
        cd3(50);              // Call M1 then M2

        cd3 -= cd2;
        cd3(60);              // Call M1

        cd3 -= cd2;           // Impossible removal is benign
        cd3(60);              // Call M1

        cd3 -= cd1;           // Invocation list is empty so cd3 is null

        //        cd3(70);    // System.NullReferenceException thrown

        cd3 -= cd1;           // Impossible removal is benign
    }
}
```


www.it-ebooks.info

As shown in the statement `cd3 += cd1;`, a delegate can be present in an invocation list multiple times. In this case, it is simply invoked once per occurrence. In such an invocation list, when that delegate is removed, the last occurrence in the invocation list is the one actually removed.

Immediately prior to the execution of the final statement `cd3 -= cd1;`, the delegate `cd3` refers to an empty invocation list. Attempting to remove a delegate from an empty list (or to remove a nonexistent delegate from a non-empty list) is not an error.

The output produced is

```
C.M1: -1
C.M2: -2
C.M1: 10
C.M2: 10
C.M1: 20
C.M2: 20
C.M1: 20
C.M1: 30
C.M2: 30
C.M1: 30
C.M3: 30
C.M1: 40
C.M2: 40
C.M3: 40
C.M1: 50
C.M2: 50
C.M1: 60
C.M1: 60
```


www.it-ebooks.info

16. Exceptions

Exceptions in C# provide a structured, uniform, and type-safe way of handling both system-level and application-level error conditions. The exception mechanism in C# is quite similar to that of C++, with a few important differences.

> **ERIC LIPPERT** The guidance on system-level and application-level exceptions used to be that all application exceptions should be derived from `ApplicationException`. This turned out to be a bad idea, and we no longer recommend this approach: Whether an exception comes from an application or the framework class library is almost always irrelevant.

- In C#, all exceptions must be represented by an instance of a class type derived from `System.Exception`. In C++, any value of any type can be used to represent an exception.

> **ERIC LIPPERT** In the first version of the CLR, it was possible to throw a non-exception and catch it in C# using the empty `catch` clause. This made for some confusing exception-handling code in C# programs that needed to catch non-exception exceptions. In newer versions of the CLR, a thrown object that is not an exception is automatically wrapped in an exception object that can be caught normally.

- In C#, a `finally` block (§8.10) can be used to write termination code that executes in both normal execution and exceptional conditions. Such code is difficult to write in C++ without duplicating code.

www.it-ebooks.info

ERIC LIPPERT One occasionally hears the myth that "the `finally` block always executes." Of course, it doesn't. The `finally` block does not execute if (1) the `try-protected` region goes into an infinite loop, (2) the program is terminated before the `finally` block runs, via a "fail fast" condition or an administrator killing the process, or (3) someone kicks the power cord out of the wall. Indeed, there are times when you do not want `finally` blocks to run. If the exception is indicative of program state being so messed up that running `finally` blocks will make the situation worse, then sometimes the right thing to do is to simply fail fast and take the process down before any more harm can be done.

- In C#, system-level exceptions such as overflow, divide-by-zero, and null dereferences have well-defined exception classes and are on a par with application-level error conditions.

BILL WAGNER This point highlights an important advantage—namely, that C# exceptions have a great deal in common with C++ exceptions. That commonality enables the C# community to leverage all the work done by the C++ community (notably Dave Abrahams and Herb Sutter) to define a set of practices that make code more robust in the face of exceptions.

CHRIS SELLS The value of having a single unified way to communicate and handle errors is one of the big advantages that .NET provides, but it's sometimes forgotten. When I see a .NET design that communicates errors in some other way, I assume it's wrong until I can be shown otherwise. I can think of only one such occasion—and that was something I designed myself, so I'm hardly an unbiased judge.

PETER SESTOFT One big exception (sorry) to Chris's rule is numeric code, where NaNs are used to report and propagate errors, and to encode information about the error in the NaN payload bits (see the annotation in §7.8.1). Throwing an exception may be three to four orders of magnitude slower than propagating a NaN. The speed is, of course, immaterial if the program must terminate with an error report even when just one exception is thrown. In some contexts, however, a numeric "error" may simply represent the absence of an input, so millions or billions of "errors" may happen during a computation; in that case, the cost matters.

www.it-ebooks.info

16.1 Causes of Exceptions

Exceptions can be thrown in two different ways.

- A `throw` statement (§8.9.5) throws an exception immediately and unconditionally. Control never reaches the statement immediately following the `throw`.
- Certain exceptional conditions that arise during the processing of C# statements and expression cause an exception in certain circumstances when the operation cannot be completed normally. For example, an integer division operation (§7.8.2) throws a `System.DivideByZeroException` if the denominator is zero. See §16.4 for a list of the various exceptions that can occur in this way.

CHRISTIAN NAGEL When throwing exceptions, you should not throw an exception of type `Exception`, but rather throw an exception of a type that derives from the `Exception` class.

16.2 The System.Exception Class

The `System.Exception` class is the base type of all exceptions. This class has a few notable properties that all exceptions share:

- `Message` is a read-only property of type `string` that contains a human-readable description of the reason for the exception.
- `InnerException` is a read-only property of type `Exception`. If its value is non-`null`, it refers to the exception that caused the current exception—that is, the current exception was raised in a `catch` block handling the `InnerException`. Otherwise, its value is `null`, indicating that this exception was not caused by another exception. The number of exception objects chained together in this manner can be arbitrary.

The value of these properties can be specified in calls to the instance constructor for `System.Exception`.

ERIC LIPPERT From this description, it sounds like exceptions are immutable objects, which they are not. Every time you throw a particular instance of an exception object, the call stack captured by that exception is reset. For example, if you create one exception instance and then throw it from multiple threads, the call stack of the exception will be observed to change on all threads that handle it, which is likely to be confusing. Create a new exception every time you need one.

www.it-ebooks.info

16.3 How Exceptions Are Handled

Exceptions are handled by a `try` statement (§8.10).

When an exception occurs, the system searches for the nearest `catch` clause that can handle the exception, as determined by the runtime type of the exception. First, the current method is searched for a lexically enclosing `try` statement, and the associated `catch` clauses of the `try` statement are considered in order. If that fails, the method that called the current method is searched for a lexically enclosing `try` statement that encloses the point of the call to the current method. This search continues until a `catch` clause is found that can handle the current exception, by naming an exception class that is of the same class, or a base class, of the runtime type of the exception being thrown. A `catch` clause that doesn't name an exception class can handle any exception.

Once a matching `catch` clause is found, the system prepares to transfer control to the first statement of the `catch` clause. Before execution of the `catch` clause begins, the system first executes, in order, any `finally` clauses that were associated with `try` statements more nested than the one that caught the exception.

If no matching `catch` clause is found, one of two things occurs:

- If the search for a matching `catch` clause reaches a static constructor (§10.12) or static field initializer, then a `System.TypeInitializationException` is thrown at the point that triggered the invocation of the static constructor. The inner exception of the `System.TypeInitializationException` contains the exception that was originally thrown.
- If the search for matching `catch` clauses reaches the code that initially started the thread, then execution of the thread is terminated. The impact of such termination is implementation-defined.

ERIC LIPPERT In earlier versions of the CLR, the implementation-defined behavior was to take down the application if the unhandled exception occurred on the main thread. If it appeared on a worker thread, then the policy was to kill the worker thread but keep running the main thread. This turned out to be a bad policy, because applications with buggy code on the worker thread would gradually lose all of their worker threads and be left with a running application that was doing no work, to the confusion of its users. The new policy is more aggressive: An unhandled exception on *any* thread takes down the application. Failing catestrophically is often a better policy than continuing on as if nothing had happened; if you fail noisily enough times, then eventually someone figures out there is a problem and fixes it.

www.it-ebooks.info

Exceptions that occur during destructor execution are worth special mention. If an exception occurs during destructor execution and that exception is not caught, then the execution of that destructor is terminated and the destructor of the base class (if any) is called. If there is no base class (as in the case of the `object` type) or if there is no base class destructor, then the exception is discarded.

> **CHRISTIAN NAGEL** The C# compiler creates a finalizer from a destructor. Within the finalizer, a `try-finally` block is added where the finalizer from the base class is called in the `finally` block.

16.4 Common Exception Classes

The following exceptions are thrown by certain C# operations.

Exception	Description
`System.ArithmeticException`	A base class for exceptions that occur during arithmetic operations, such as `System.DivideByZeroException` and `System.OverflowException`.
`System.ArrayTypeMismatchException`	Thrown when a store into an array fails because the actual type of the stored element is incompatible with the actual type of the array.
`System.DivideByZeroException`	Thrown when an attempt to divide an integral value by zero occurs.
`System.IndexOutOfRangeException`	Thrown when an attempt to index an array via an index that is less than zero or outside the bounds of the array occurs.
`System.InvalidCastException`	Thrown when an explicit conversion from a base type or interface to a derived type fails at runtime.
`System.NullReferenceException`	Thrown when a `null` reference is used in a way that causes the referenced object to be required.
`System.OutOfMemoryException`	Thrown when an attempt to allocate memory (via `new`) fails.

Continued

www.it-ebooks.info

Exception	Description
`System.OverflowException`	Thrown when an arithmetic operation in a `checked` context overflows.
`System.StackOverflowException`	Thrown when the execution stack is exhausted by having too many pending method calls; typically indicative of very deep or unbounded recursion.
`System.TypeInitializationException`	Thrown when a static constructor throws an exception, and no `catch` clauses exists to catch it.

JON SKEET This table does not list the most commonly encountered exceptions, but rather the exceptions that can occur naturally within the execution environment as the result of C# operations. You should very rarely (if ever) explicitly throw these exceptions yourself. By contrast, the more commonly encountered exceptions such as `System.ArgumentException` and its subclasses aren't listed here—they're thrown by a higher level of code, whether system libraries or application code.

ERIC LIPPERT I categorize exceptions as *fatal*, *boneheaded*, *vexing*, and *exogenous*.

Fatal exceptions you neither throw nor catch; they include out of memory, thread abort, and so on. You did not cause the problem and you cannot fix it; the exception is just the mechanism by which you are notified that the world is about to end.

Boneheaded exceptions are avoidable and, therefore, are thrown only by buggy programs; they include null dereferenced, invalid argument, and so on. Never catch them; catching a boneheaded exception is hiding someone's bug. Instead, fix the bug.

Vexing exceptions are exceptions that you have to handle because the API is designed to communicate facts via exceptions. For example, previous versions of the framework lacked an "Is this string a legal GUID?" method; to answer that question, you might try to call the GUID constructor and catch the vexing exception to see if it succeeded or failed. Try to not write code that throws vexing exceptions.

Exogenous exceptions are those you must catch because they indicate unexpected facts about the real world: the CD has been removed from the drive, the network router has been unplugged, and so on.

www.it-ebooks.info

17. Attributes

Much of the C# language enables the programmer to specify declarative information about the entities defined in the program. For example, the accessibility of a method in a class is specified by decorating it with the *method-modifiers* `public`, `protected`, `internal`, and `private`.

C# enables programmers to invent new kinds of declarative information, called ***attributes***. Programmers can then attach attributes to various program entities, and retrieve attribute information in a runtime environment. For instance, a framework might define a `HelpAttribute` attribute that can be placed on certain program elements (such as classes and methods) to provide a mapping from those program elements to their documentation.

> **JESSE LIBERTY** A common and powerful use of attributes can be seen throughout Silverlight and WPF, where attributes are used to indicate the available view states associated with a class.
>
> Similarly, Test Libraries use attributes to differentiate test methods from supporting methods. Web services use attributes to designate which methods are exposed to clients.

Attributes are defined through the declaration of attribute classes (§17.1), which may have positional and named parameters (§17.1.2). Attributes are attached to entities in a C# program using attribute specifications (§17.2), and can be retrieved at runtime as attribute instances (§17.3).

> **ERIC LIPPERT** Try to use attributes only to talk about *the type itself* rather than representing details about the *semantics* of that type. For example, suppose you have a class `Book` with a property `Author`. That is part of the semantics of the class: The class represents books, and books have authors. If you put an `AuthorAttribute` on the class `Book`, that doesn't represent the author of the *book,* but rather the author of the *class.* A class `Television` might have a property `Obsolete` that indicates whether a particular model is out of production. If you put an `ObsoleteAttribute` on the class, it means that *the class itself* is obsolete, not that it *represents an obsolete product.*

www.it-ebooks.info

17.1 Attribute Classes

A class that derives from the abstract class `System.Attribute`, whether directly or indirectly, is an ***attribute class***. The declaration of an attribute class defines a new kind of ***attribute*** that can be placed on a declaration. By convention, attribute classes are named with a suffix of `Attribute`. Uses of an attribute may either include or omit this suffix.

17.1.1 Attribute Usage

The attribute `AttributeUsage` (§17.4.1) is used to describe how an attribute class can be used.

`AttributeUsage` has a positional parameter (§17.1.2) that enables an attribute class to specify the kinds of declarations on which it can be used. The example

```
using System;

[AttributeUsage(AttributeTargets.Class | AttributeTargets.Interface)]
public class SimpleAttribute: Attribute
{
    ...
}
```

defines an attribute class named `SimpleAttribute` that can be placed on *class-declaration*s and *interface-declaration*s only. The example

```
[Simple] class Class1 {...}
[Simple] interface Interface1 {...}
```

shows several uses of the `Simple` attribute. Although this attribute is defined with the name `SimpleAttribute`, when this attribute is used, the `Attribute` suffix may be omitted, resulting in the short name `Simple`. Thus the example above is semantically equivalent to the following:

```
[SimpleAttribute] class Class1 {...}
[SimpleAttribute] interface Interface1 {...}
```

`AttributeUsage` has a named parameter (§17.1.2) called `AllowMultiple`, which indicates whether the attribute can be specified more than once for a given entity. If `AllowMultiple` for an attribute class is `true`, then that attribute class is a ***multi-use attribute class***, and can be specified more than once on an entity. If `AllowMultiple` for an attribute class is `false` or it is unspecified, then that attribute class is a ***single-use attribute class***, and can be specified at most once on an entity.

www.it-ebooks.info

The example

```
using System;

[AttributeUsage(AttributeTargets.Class, AllowMultiple = true)]
public class AuthorAttribute: Attribute
{
    private string name;

    public AuthorAttribute(string name) {
        this.name = name;
    }

    public string Name {
        get { return name; }
    }
}
```

defines a multi-use attribute class named `AuthorAttribute`. The example

```
[Author("Brian Kernighan"), Author("Dennis Ritchie")]
class Class1
{
    ...
}
```

shows a class declaration with two uses of the `Author` attribute.

CHRIS SELLS For those of you who are reading this example and don't know who Brian Kernighan or Dennis Ritchie is: For shame! They are true software legends.

JON SKEET While I agree with Chris, I certainly hope that neither of the esteemed gentlemen in question would ever write a class named `Class1`.

`AttributeUsage` has another named parameter called `Inherited`, which indicates whether the attribute, when specified on a base class, is also inherited by classes that derive from that base class. If `Inherited` for an attribute class is `true`, then that attribute is inherited. If `Inherited` for an attribute class is `false`, then that attribute is not inherited. If it is unspecified, its default value is `true`.

An attribute class `X` not having an `AttributeUsage` attribute attached to it, as in

```
using System;

class X: Attribute {...}
```


www.it-ebooks.info

is equivalent to the following:

```
using System;

[AttributeUsage(
    AttributeTargets.All,
    AllowMultiple = false,
    Inherited = true)
]
class X: Attribute {...}
```

17.1.2 Positional and Named Parameters

> **BILL WAGNER** In most cases, using named parameters will result in a clearer, more easily understood program. Positional parameters should be used only for attributes with one property, where the property is made clear by the attribute name, such as `AuthorAttribute`.

Attribute classes can have ***positional parameters*** and ***named parameters***. Each public instance constructor for an attribute class defines a valid sequence of positional parameters for that attribute class. Each nonstatic public read-write field and property for an attribute class defines a named parameter for the attribute class.

The example

```
using System;

[AttributeUsage(AttributeTargets.Class)]
public class HelpAttribute: Attribute
{
    public HelpAttribute(string url) {  // Positional parameter
        ...
    }

    public string Topic {               // Named parameter
        get {...}
        set {...}
    }

    public string Url {
        get {...}
    }
}
```

defines an attribute class named `HelpAttribute` that has one positional parameter, `url`, and one named parameter, `Topic`. Although it is nonstatic and public, the property `Url` does not define a named parameter, since it is not read-write.

www.it-ebooks.info

This attribute class might be used as follows:

```
[Help("http://www.mycompany.com/.../Class1.htm")]
class Class1
{
    ...
}

[Help("http://www.mycompany.com/.../Misc.htm", Topic = "Class2")]
class Class2
{
    ...
}
```

17.1.3 Attribute Parameter Types

The types of positional and named parameters for an attribute class are limited to the ***attribute parameter types***:

- One of the following types: `bool`, `byte`, `char`, `double`, `float`, `int`, `long`, `sbyte`, `short`, `string`, `uint`, `ulong`, `ushort`.
- The type `object`.
- The type `System.Type`.
- An enum type, provided it has public accessibility and the types in which it is nested (if any) also have public accessibility (§17.2).
- Single-dimensional arrays of the above types.

JON SKEET Note that `decimal` cannot be used as a parameter type, even though you can declare `decimal` constants. This is one example of CLI rules leaking into C#.

MAREK SAFAR Although arrays are supported, array covariance is not allowed for attributes of reference types. Similarly, the C# 4.0 `dynamic` type cannot be used where type `object` is expected.

A constructor argument or public field that does not have one of these types cannot be used as a positional or named parameter in an attribute specification.

www.it-ebooks.info

17.2 Attribute Specification

Attribute specification is the application of a previously defined attribute to a declaration. An attribute is a piece of additional declarative information that is specified for a declaration. Attributes can be specified at global scope (to specify attributes on the containing assembly or module) and for *type-declaration*s (§9.6), *class-member-declaration*s (§10.1.5), *interface-member-declaration*s (§13.2), *struct-member-declaration*s (§11.2), *enum-member-declaration*s (§14.3), *accessor-declaration*s (§10.7.2), *event-accessor-declarations* (§10.8.1), and *formal-parameter-lists* (§10.6.1).

Attributes are specified in ***attribute sections***. An attribute section consists of a pair of square brackets, which surround a comma-separated list of one or more attributes. The order in which attributes are specified in such a list, and the order in which sections attached to the same program entity are arranged, is not significant. For instance, the attribute specifications `[A][B]`, `[B][A]`, `[A, B]`, and `[B, A]` are all equivalent.

global-attributes:
    global-attribute-sections

global-attribute-sections:
    global-attribute-section
    global-attribute-sections *global-attribute-section*

global-attribute-section:
    `[` *global-attribute-target-specifier* *attribute-list* `]`
    `[` *global-attribute-target-specifier* *attribute-list* `,` `]`

global-attribute-target-specifier:
    global-attribute-target `:`

global-attribute-target:
    `assembly`
    `module`

attributes:
    attribute-sections

attribute-sections:
    attribute-section
    attribute-sections *attribute-section*

attribute-section:
    `[` *attribute-target-specifier*$_{opt}$ *attribute-list* `]`
    `[` *attribute-target-specifier*$_{opt}$ *attribute-list* `,` `]`

www.it-ebooks.info

attribute-target-specifier:
    attribute-target `:`

attribute-target:
    `field`
    `event`
    `method`
    `param`
    `property`
    `return`
    `type`

attribute-list:
    attribute
    attribute-list `,` *attribute*

attribute:
    attribute-name *attribute-arguments*$_{opt}$

attribute-name:
    type-name

attribute-arguments:
    `(` *positional-argument-list*$_{opt}$ `)`
    `(` *positional-argument-list* `,` *named-argument-list* `)`
    `(` *named-argument-list* `)`

positional-argument-list:
    positional-argument
    positional-argument-list `,` *positional-argument*

positional-argument:
    argument-name$_{opt}$ *attribute-argument-expression*

named-argument-list:
    named-argument
    named-argument-list `,` *named-argument*

named-argument:
    identifier `=` *attribute-argument-expression*

attribute-argument-expression:
    expression

www.it-ebooks.info

An attribute consists of an *attribute-name* and an optional list of positional and named arguments. The positional arguments (if any) precede the named arguments. A positional argument consists of an *attribute-argument-expression*; a named argument consists of a name, followed by an equal sign, followed by an *attribute-argument-expression*, which together are constrained by the same rules as simple assignment. The order of named arguments is not significant.

The *attribute-name* identifies an attribute class. If the form of *attribute-name* is *type-name*, then this name must refer to an attribute class. Otherwise, a compile-time error occurs. The example

```
class Class1 {}

[Class1] class Class2 {}     // Error
```

results in a compile-time error because it attempts to use `Class1` as an attribute class when `Class1` is not an attribute class.

Certain contexts permit the specification of an attribute on more than one target. A program can explicitly specify the target by including an *attribute-target-specifier*. When an attribute is placed at the global level, a *global-attribute-target-specifier* is required. In all other locations, a reasonable default is applied, but an *attribute-target-specifier* can be used to affirm or override the default in certain ambiguous cases (or to just affirm the default in non-ambiguous cases). Thus, typically, *attribute-target-specifier*s can be omitted except at the global level. The potentially ambiguous contexts are resolved as follows:

- An attribute specified at the global scope can apply either to the target assembly or the target module. No default exists for this context, so an *attribute-target-specifier* is always required in this context. The presence of the `assembly` *attribute-target-specifier* indicates that the attribute applies to the target assembly; the presence of the `module` *attribute-target-specifier* indicates that the attribute applies to the target module.
- An attribute specified on a delegate declaration can apply either to the delegate being declared or to its return value. In the absence of an *attribute-target-specifier*, the attribute applies to the delegate. The presence of the `type` *attribute-target-specifier* indicates that the attribute applies to the delegate; the presence of the `return` *attribute-target-specifier* indicates that the attribute applies to the return value.
- An attribute specified on a method declaration can apply either to the method being declared or to its return value. In the absence of an *attribute-target-specifier*, the attribute applies to the method. The presence of the `method` *attribute-target-specifier* indicates that the attribute applies to the method; the presence of the `return` *attribute-target-specifier* indicates that the attribute applies to the return value.

www.it-ebooks.info

- An attribute specified on an operator declaration can apply either to the operator being declared or to its return value. In the absence of an *attribute-target-specifier*, the attribute applies to the operator. The presence of the `method` *attribute-target-specifier* indicates that the attribute applies to the operator; the presence of the `return` *attribute-target-specifier* indicates that the attribute applies to the return value.
- An attribute specified on an event declaration that omits event accessors can apply to the event being declared, to the associated field (if the event is not abstract), or to the associated add and remove methods. In the absence of an *attribute-target-specifier*, the attribute applies to the event. The presence of the `event` *attribute-target-specifier* indicates that the attribute applies to the event; the presence of the `field` *attribute-target-specifier* indicates that the attribute applies to the field; and the presence of the `method` *attribute-target-specifier* indicates that the attribute applies to the methods.
- An attribute specified on a `get` accessor declaration for a property or indexer declaration can apply either to the associated method or to its return value. In the absence of an *attribute-target-specifier*, the attribute applies to the method. The presence of the `method` *attribute-target-specifier* indicates that the attribute applies to the method; the presence of the `return` *attribute-target-specifier* indicates that the attribute applies to the return value.
- An attribute specified on a `set` accessor for a property or indexer declaration can apply either to the associated method or to its lone implicit parameter. In the absence of an *attribute-target-specifier*, the attribute applies to the method. The presence of the `method` *attribute-target-specifier* indicates that the attribute applies to the method; the presence of the `param` *attribute-target-specifier* indicates that the attribute applies to the parameter; and the presence of the `return` *attribute-target-specifier* indicates that the attribute applies to the return value.
- An attribute specified on an add or remove accessor declaration for an event declaration can apply either to the associated method or to its lone parameter. In the absence of an *attribute-target-specifier*, the attribute applies to the method. The presence of the `method` *attribute-target-specifier* indicates that the attribute applies to the method; the presence of the `param` *attribute-target-specifier* indicates that the attribute applies to the parameter; and the presence of the `return` *attribute-target-specifier* indicates that the attribute applies to the return value.

In other contexts, inclusion of an *attribute-target-specifier* is permitted but unnecessary. For instance, a class declaration may either include or omit the specifier `type`:

```
[type: Author("Brian Kernighan")]
class Class1 {}

[Author("Dennis Ritchie")]
class Class2 {}
```


www.it-ebooks.info

It is an error to specify an invalid *attribute-target-specifier*. For instance, the specifier `param` cannot be used on a class declaration:

```
[param: Author("Brian Kernighan")]        // Error
class Class1 {}
```

By convention, attribute classes are named with a suffix of `Attribute`. An *attribute-name* of the form *type-name* may either include or omit this suffix. If an attribute class is found both with and without this suffix, an ambiguity is present, and a compile-time error results. If the *attribute-name* is spelled such that its rightmost *identifier* is a verbatim identifier (§2.4.2), then only an attribute without a suffix is matched, thus enabling such an ambiguity to be resolved. The example

```
using System;

[AttributeUsage(AttributeTargets.All)]
public class X: Attribute
{}

[AttributeUsage(AttributeTargets.All)]
public class XAttribute: Attribute
{}

[X]                           // Error: ambiguity
class Class1 {}

[XAttribute]                  // Refers to XAttribute
class Class2 {}

[@X]                          // Refers to X
class Class3 {}

[@XAttribute]                 // Refers to XAttribute
class Class4 {}
```

shows two attribute classes named `X` and `XAttribute`. The attribute `[X]` is ambiguous, since it could refer to either `X` or `XAttribute`. Using a verbatim identifier allows the exact intent to be specified in such rare cases. The attribute `[XAttribute]` is not ambiguous (although it would be if there was an attribute class named `XAttributeAttribute`!). If the declaration for class `X` is removed, then both attributes refer to the attribute class named `XAttribute`, as follows:

```
using System;

[AttributeUsage(AttributeTargets.All)]
public class XAttribute: Attribute
{}

[X]                           // Refers to XAttribute
class Class1 {}

[XAttribute]                  // Refers to XAttribute
class Class2 {}

[@X]                          // Error: no attribute named "X"
class Class3 {}
```


www.it-ebooks.info

It is a compile-time error to use a single-use attribute class more than once on the same entity. The example

```
using System;

[AttributeUsage(AttributeTargets.Class)]
public class HelpStringAttribute: Attribute
{
    string value;

    public HelpStringAttribute(string value) {
        this.value = value;
    }

    public string Value {
        get {...}
    }
}

[HelpString("Description of Class1")]
[HelpString("Another description of Class1")]
public class Class1 {}
```

results in a compile-time error because it attempts to use `HelpString`, which is a single-use attribute class, more than once on the declaration of `Class1`.

An expression `E` is an *attribute-argument-expression* if all of the following statements are true:

- The type of `E` is an attribute parameter type (§17.1.3).
- At compile time, the value of `E` can be resolved to one of the following:
  - A constant value.
  - A `System.Type` object.
  - A one-dimensional array of *attribute-argument-expression*s.

For example:

```
using System;

[AttributeUsage(AttributeTargets.Class)]
public class TestAttribute: Attribute
{
    public int P1 {
        get {...}
        set {...}
    }

    public Type P2 {
        get {...}
        set {...}
    }
```


www.it-ebooks.info

```
    public object P3 {
        get {...}
        set {...}
    }
}

[Test(P1 = 1234, P3 = new int[] {1, 3, 5}, P2 = typeof(float))]
class MyClass {}
```

A *typeof-expression* (§7.6.11) used as an attribute argument expression can reference a non-generic type, a closed constructed type, or an unbound generic type, but it cannot reference an open type. This is to ensure that the expression can be resolved at compile time.

```
class A: Attribute
{
    public A(Type t) {...}
}

class G<T>
{
    [A(typeof(T))] T t;                 // Error: open type in attribute
}

class X
{
    [A(typeof(List<int>))] int x;       // Okay: closed constructed type
    [A(typeof(List<>))] int y;          // Okay: unbound generic type
}
```

17.3 Attribute Instances

An ***attribute instance*** is an instance that represents an attribute at runtime. An attribute is defined with an attribute class, positional arguments, and named arguments. An attribute instance is an instance of the attribute class that is initialized with the positional and named arguments.

Retrieval of an attribute instance involves both compile-time and runtime processing, as described in the following sections.

17.3.1 Compilation of an Attribute

The compilation of an *attribute* with attribute class `T`, *positional-argument-list* `P`, and *named-argument-list* `N`, consists of the following steps:

- Follow the compile-time processing steps for compiling an *object-creation-expression* of the form `new T(P)`. These steps either result in a compile-time error or determine an instance constructor `C` on `T` that can be invoked at runtime.
- If `C` does not have public accessibility, then a compile-time error occurs.

www.it-ebooks.info

- For each *named-argument* `Arg` in `N`:
  - Let `Name` be the *identifier* of the *named-argument* `Arg`.
  - `Name` must identify a nonstatic read-write public field or property on `T`. If `T` has no such field or property, then a compile-time error occurs.
- Keep the following information for runtime instantiation of the attribute: the attribute class `T`, the instance constructor `C` on `T`, the *positional-argument-list* `P`, and the *named-argument-list* `N`.

MAREK SAFAR The public accessibility restriction is a little bit confusing in this case. For instance, class attributes are declared lexically outside the class but they can still use private constants declared within the same class.

17.3.2 Runtime Retrieval of an Attribute Instance

Compilation of an *attribute* yields an attribute class `T`, an instance constructor `C` on `T`, a *positional-argument-list* `P`, and a *named-argument-list* `N`. Given this information, an attribute instance can be retrieved at runtime using the following steps:

- Follow the runtime processing steps for executing an *object-creation-expression* of the form `new T(P)`, using the instance constructor `C` as determined at compile time. These steps either result in an exception, or produce an instance `O` of `T`.
- For each *named-argument* `Arg` in `N`, in order:
  - Let `Name` be the *identifier* of the *named-argument* `Arg`. If `Name` does not identify a non-static public read-write field or property on `O`, then an exception is thrown.
  - Let `Value` be the result of evaluating the *attribute-argument-expression* of `Arg`.
  - If `Name` identifies a field on `O`, then set this field to `Value`.
  - Otherwise, `Name` identifies a property on `O`. Set this property to `Value`.
  - The result is `O`, an instance of the attribute class `T` that has been initialized with the *positional-argument-list* `P` and the *named-argument-list* `N`.

17.4 Reserved Attributes

A small number of attributes affect the language in some way. These attributes include:

- `System.AttributeUsageAttribute` (§17.4.1), which is used to describe the ways in which an attribute class can be used.

www.it-ebooks.info

- `System.Diagnostics.ConditionalAttribute` (§17.4.2), which is used to define conditional methods.
- `System.ObsoleteAttribute` (§17.4.3), which is used to mark a member as obsolete.

17.4.1 The AttributeUsage Attribute

The attribute `AttributeUsage` is used to describe the manner in which the attribute class can be used.

A class that is decorated with the `AttributeUsage` attribute must derive from `System.Attribute`, either directly or indirectly. Otherwise, a compile-time error occurs.

```
namespace System
{
    [AttributeUsage(AttributeTargets.Class)]
    public class AttributeUsageAttribute: Attribute
    {
        public AttributeUsageAttribute(AttributeTargets validOn) {
                ...
            }

        public virtual bool AllowMultiple { get {...} set {...} }

        public virtual bool Inherited { get {...} set {...} }

        public virtual AttributeTargets ValidOn { get {...} }
    }

    public enum AttributeTargets
    {
        Assembly    = 0x0001,
        Module      = 0x0002,
        Class       = 0x0004,
        Struct      = 0x0008,
        Enum        = 0x0010,
        Constructor = 0x0020,
        Method      = 0x0040,
        Property    = 0x0080,
        Field       = 0x0100,
        Event       = 0x0200,
        Interface   = 0x0400,
        Parameter   = 0x0800,
        Delegate    = 0x1000,
        ReturnValue = 0x2000,

        All = Assembly | Module | Class | Struct | Enum |
            Constructor | Method | Property | Field | Event |
            Interface | Parameter | Delegate | ReturnValue
    }
}
```


www.it-ebooks.info

17.4.2 The Conditional Attribute

The attribute `Conditional` enables the definition of ***conditional methods*** and ***conditional attribute classes***.

```
namespace System.Diagnostics
{
    [AttributeUsage(AttributeTargets.Method | AttributeTargets.Class,
             AllowMultiple = true)]
    public class ConditionalAttribute: Attribute
    {
        public ConditionalAttribute(string conditionString) {...}

        public string ConditionString { get {...} }
    }
}
```

17.4.2.1 *Conditional Methods*

A method decorated with the `Conditional` attribute is a conditional method. The `Conditional` attribute indicates a condition by testing a conditional compilation symbol. Calls to a conditional method are either included or omitted depending on whether this symbol is defined at the point of the call. If the symbol is defined, the call is included; otherwise, the call (including evaluation of the receiver and parameters of the call) is omitted.

A conditional method is subject to the following restrictions:

- The conditional method must be a method in a *class-declaration* or *struct-declaration*. A compile-time error occurs if the `Conditional` attribute is specified on a method in an interface declaration.
- The conditional method must have a return type of `void`.
- The conditional method must not be marked with the `override` modifier. A conditional method may be marked with the `virtual` modifier, however. Overrides of such a method are implicitly conditional, and must not be explicitly marked with a `Conditional` attribute.
- The conditional method must not be an implementation of an interface method. Otherwise, a compile-time error occurs.

In addition, a compile-time error occurs if a conditional method is used in a *delegate-creation-expression*. The example

```
#define DEBUG

using System;
using System.Diagnostics;
class Class1
```


www.it-ebooks.info

```
{
    [Conditional("DEBUG")]
    public static void M() {
        Console.WriteLine("Executed Class1.M");
    }
}

class Class2
{
    public static void Test() {
        Class1.M();
    }
}
```

declares `Class1.M` as a conditional method. `Class2`'s `Test` method calls this method. Since the conditional compilation symbol `DEBUG` is defined, if `Class2.Test` is called, it will call `M`. If the symbol `DEBUG` had not been defined, then `Class2.Test` would not call `Class1.M`.

It is important to note that the inclusion or exclusion of a call to a conditional method is controlled by the conditional compilation symbols at the point of the call. In the example

File `class1.cs`:

```
using System.Diagnostics;

class Class1
{
    [Conditional("DEBUG")]
    public static void F() {
        Console.WriteLine("Executed Class1.F");
    }
}
```

File `class2.cs`:

```
#define DEBUG

class Class2
{
    public static void G() {
        Class1.F();                 // F is called
    }
}
```

File `class3.cs`:

```
#undef DEBUG

class Class3
{
    public static void H() {
        Class1.F();                 // F is not called
    }
}
```


www.it-ebooks.info

the classes `Class2` and `Class3` each contain calls to the conditional method `Class1.F`, which is conditional based on whether `DEBUG` is defined. Since this symbol is defined in the context of `Class2` but not `Class3`, the call to `F` in `Class2` is included, while the call to `F` in `Class3` is omitted.

The use of conditional methods in an inheritance chain can be confusing. Calls made to a conditional method through `base`, of the form `base.M`, are subject to the normal conditional method call rules. In the example

File `class1.cs`:

```
using System;
using System.Diagnostics;

class Class1
{
    [Conditional("DEBUG")]
    public virtual void M() {
        Console.WriteLine("Class1.M executed");
    }
}
```

File `class2.cs`:

```
using System;

class Class2: Class1
{
    public override void M() {
        Console.WriteLine("Class2.M executed");
        base.M();               // base.M is not called!
    }
}
```

File `class3.cs`:

```
#define DEBUG

using System;

class Class3
{
    public static void Test() {
        Class2 c = new Class2();
        c.M();                  // M is called
    }
}
```

`Class2` includes a call to the `M` defined in its base class. This call is omitted because the base method is conditional based on the presence of the symbol `DEBUG`, which is undefined. Thus the method writes to the console "`Class2.M executed`" only. Judicious use of *pp-declaration*s can eliminate such problems.

www.it-ebooks.info

17.4.2.2 *Conditional Attribute Classes*

An attribute class (§17.1) decorated with one or more `Conditional` attributes is a ***conditional attribute class***. A conditional attribute class is thus associated with the conditional compilation symbols declared in its `Conditional` attributes. The example

```
using System;
using System.Diagnostics;
[Conditional("ALPHA")]
[Conditional("BETA")]
public class TestAttribute : Attribute {}
```

declares `TestAttribute` as a conditional attribute class associated with the conditional compilations symbols `ALPHA` and `BETA`.

Attribute specifications (§17.2) of a conditional attribute are included if one or more of its associated conditional compilation symbols are defined at the point of specification; otherwise, the attribute specification is omitted.

It is important to note that the inclusion or exclusion of an attribute specification of a conditional attribute class is controlled by the conditional compilation symbols at the point of the specification. In the example

File `test.cs`:

```
using System;
using System.Diagnostics;

[Conditional("DEBUG")]
public class TestAttribute : Attribute {}
```

File `class1.cs`:

```
#define DEBUG

[Test]                          // TestAttribute is specified
class Class1 {}
```

File `class2.cs`:

```
#undef DEBUG

[Test]                           // TestAttribute is not specified
class Class2 {}
```

the classes `Class1` and `Class2` are each decorated with attribute `Test`, which is conditional based on whether `DEBUG` is defined. Since this symbol is defined in the context of `Class1` but not `Class2`, the specification of the `Test` attribute on `Class1` is included, while the specification of the `Test` attribute on `Class2` is omitted.

www.it-ebooks.info

ERIC LIPPERT The conditional attribute clearly has a strong connection to conditional compilation symbols—but it is important to remember that they are very different. A common error is to write something like this:

```
#if DEBUG
int counter;
#endif
[Conditional("DEBUG")] void DoIt(int x) { ... }
...
DoIt(this.counter);
```

The definition of the field `counter` is removed from the program entirely if the `DEBUG` symbol is not defined. How does the compiler know to remove the call to `DoIt`? Because `DoIt(int)` is conditional and `DEBUG` is not defined. But how does the compiler know that `DoIt(int)` is being called, as opposed to some other overload? Because this counter is declared as an `int`—oh, wait, *that field definition has been removed.* There is no such field, so compilation will fail in the non-debug build.

17.4.3 The Obsolete Attribute

The attribute `Obsolete` is used to mark types and members of types that should no longer be used.

```
namespace System
{
    [AttributeUsage(
        AttributeTargets.Class |
        AttributeTargets.Struct |
        AttributeTargets.Enum |
        AttributeTargets.Interface |
        AttributeTargets.Delegate |
        AttributeTargets.Method |
        AttributeTargets.Constructor |
        AttributeTargets.Property |
        AttributeTargets.Field |
        AttributeTargets.Event,
        Inherited = false)
    ]
    public class ObsoleteAttribute : Attribute
    {
        public ObsoleteAttribute() {...}

        public ObsoleteAttribute(string message) {...}

        public ObsoleteAttribute(string message, bool error) {...}

        public string Message { get {...} }

        public bool IsError { get {...} }
    }
}
```


www.it-ebooks.info

If a program uses a type or member that is decorated with the `Obsolete` attribute, the compiler issues a warning or an error. Specifically, the compiler issues a warning if no error parameter is provided, or if the error parameter is provided and has the value `false`. The compiler issues an error if the error parameter is specified and has the value `true`.

In the example

```
[Obsolete("This class is obsolete; use class B instead")]
class A
{
    public void F() { }
}

class B
{
    public void F() { }
}

class Test
{
    static void Main()
    {
        A a = new A();        // Warning
        a.F();
    }
}
```

the class `A` is decorated with the `Obsolete` attribute. Each use of `A` in `Main` results in a warning that includes the specified message, "This class is obsolete; use class B instead."

JON SKEET There's an important lesson in this example: When you make something obsolete, always give guidance as to the new, preferred way of achieving a similar effect. Ideally, provide some indication (potentially in the documentation instead of the attribute message) of why the "old" code is being declared obsolete as well. It's extremely annoying when code that has worked for a long time suddenly starts spouting warnings with no good explanation.

MAREK SAFAR The compiler does not automatically obsolete overrides of virtual methods. Therefore, when a virtual method is marked as `Obsolete`, every method that overrides it has to be manually decorated with the `Obsolete` attribute as well.

www.it-ebooks.info

17.5 Attributes for Interoperation

Note: This section is applicable only to the Microsoft .NET implementation of C#.

17.5.1 Interoperation with COM and Win32 Components

The .NET runtime provides a large number of attributes that enable C# programs to interoperate with components written using COM and Win32 DLLs. For example, the `DllImport` attribute can be used on a `static extern` method to indicate that the implementation of the method is to be found in a Win32 DLL. These attributes are found in the `System.Runtime.InteropServices` namespace, and detailed documentation for these attributes is found in the .NET runtime documentation.

17.5.2 Interoperation with Other .NET Languages

17.5.2.1 *The IndexerName Attribute*

Indexers are implemented in .NET using indexed properties, and have a name in the .NET metadata. If no `IndexerName` attribute is present for an indexer, then the name `Item` is used by default. The `IndexerName` attribute enables a developer to override this default and specify a different name.

```
namespace System.Runtime.CompilerServices.CSharp
{
    [AttributeUsage(AttributeTargets.Property)]
    public class IndexerNameAttribute : Attribute
    {
        public IndexerNameAttribute(string indexerName) {...}

        public string Value { get {...} }
    }
}
```


www.it-ebooks.info

This page intentionally left blank

www.it-ebooks.info

18. Unsafe Code

The core C# language, as defined in the preceding chapters, differs notably from C and C++ in its omission of pointers as a data type. Instead, C# provides references and the ability to create objects that are managed by a garbage collector. This design, coupled with other features, makes C# a much safer language than C or C++. In the core C# language, it is simply not possible to have an uninitialized variable, a "dangling" pointer, or an expression that indexes an array beyond its bounds. Whole categories of bugs that routinely plague C and C++ programs are thus eliminated.

While practically every pointer type construct in C or C++ has a reference type counterpart in C#, there are still some situations in which access to pointer types becomes a necessity. For example, interfacing with the underlying operating system, accessing a memory-mapped device, or implementing a time-critical algorithm may not be possible or practical without access to pointers. To address this need, C# provides the ability to write ***unsafe code***.

> **CHRIS SELLS** In all the time since I've been programming .NET (starting before the RTM of 1.0), I've never once found a need to write unsafe code. Not once.

In unsafe code, it is possible to declare and operate on pointers, to perform conversions between pointers and integral types, to take the address of variables, and so forth. In a sense, writing unsafe code is much like writing C code within a C# program.

Unsafe code is, in fact, a "safe" feature from the perspective of both developers and users. Unsafe code must be clearly marked with the modifier `unsafe`, so developers cannot possibly use unsafe features accidentally, and the execution engine works to ensure that unsafe code cannot be executed in an untrusted environment.

www.it-ebooks.info

18.1 Unsafe Contexts

The unsafe features of C# are available only in *unsafe contexts*. An unsafe context is introduced by including an `unsafe` modifier in the declaration of a type or member or by employing an *unsafe-statement*:

- A declaration of a class, struct, interface, or delegate may include an `unsafe` modifier, in which case the entire textual extent of that type declaration (including the body of the class, struct, or interface) is considered an unsafe context.
- A declaration of a field, method, property, event, indexer, operator, instance constructor, destructor, or static constructor may include an `unsafe` modifier, in which case the entire textual extent of that member declaration is considered an unsafe context.
- An *unsafe-statement* enables the use of an unsafe context within a *block*. The entire textual extent of the associated *block* is considered an unsafe context.

The associated grammar extensions are shown below. For brevity, ellipses (...) are used to represent productions that appear in preceding chapters.

class-modifier:
    ...
    `unsafe`

struct-modifier:
    ...
    `unsafe`

interface-modifier:
    ...
    `unsafe`

delegate-modifier:
    ...
    `unsafe`

field-modifier:
    ...
    `unsafe`

method-modifier:
    ...
    `unsafe`

property-modifier:
    ...
    `unsafe`

www.it-ebooks.info

event-modifier:
    ...
    `unsafe`

indexer-modifier:
    ...
    `unsafe`

operator-modifier:
    ...
    `unsafe`

constructor-modifier:
    ...
    `unsafe`

destructor-declaration:
    attributes$_{opt}$ `extern`$_{opt}$ `unsafe`$_{opt}$ `~` *identifier* `(` `)` *destructor-body*
    attributes$_{opt}$ `unsafe`$_{opt}$ `extern`$_{opt}$ `~` *identifier* `(` `)` *destructor-body*

static-constructor-modifiers:
    `extern`$_{opt}$ `unsafe`$_{opt}$ `static`
    `unsafe`$_{opt}$ `extern`$_{opt}$ `static`
    `extern`$_{opt}$ `static` `unsafe`$_{opt}$
    `unsafe`$_{opt}$ `static` `extern`$_{opt}$
    `static` `extern`$_{opt}$ `unsafe`$_{opt}$
    `static` `unsafe`$_{opt}$ `extern`$_{opt}$

embedded-statement:
    ...
    unsafe-statement

unsafe-statement:
    `unsafe` *block*

In the example

```
public unsafe struct Node
{
    public int Value;
    public Node* Left;
    public Node* Right;
}
```


www.it-ebooks.info

the `unsafe` modifier specified in the struct declaration causes the entire textual extent of the struct declaration to become an unsafe context. Thus it is possible to declare the `Left` and `Right` fields to be of a pointer type. The example above could also be written as follows:

```
public struct Node
{
    public int Value;
    public unsafe Node* Left;
    public unsafe Node* Right;
}
```

Here, the `unsafe` modifiers in the field declarations cause those declarations to be considered unsafe contexts.

Other than establishing an unsafe context, thereby permitting the use of pointer types, the `unsafe` modifier has no effect on a type or a member. In the example

```
public class A
{
    public unsafe virtual void F() {
           char* p;
           ...
      }
}

public class B : A
{
    public override void F() {
           base.F();
           ...
      }
}
```

the `unsafe` modifier on the `F` method in `A` simply causes the textual extent of `F` to become an unsafe context in which the unsafe features of the language can be used. In the override of `F` in `B`, there is no need to re-specify the `unsafe` modifier—unless, of course, the `F` method in `B` itself needs access to unsafe features.

The situation is slightly different when a pointer type is part of the method's signature:

```
public unsafe class A
{
    public virtual void F(char* p) {...}
}

public class B : A
{
    public unsafe override void F(char* p) {...}
}
```

Here, because `F`'s signature includes a pointer type, it can be written only in an unsafe context. However, the unsafe context can be introduced either by making the entire class

www.it-ebooks.info

unsafe, as is the case in `A`, or by including an `unsafe` modifier in the method declaration, as is the case in `B`.

18.2 Pointer Types

In an unsafe context, a *type* (§4) may be a *pointer-type* as well as a *value-type* or a *reference-type*. However, a *pointer-type* may also be used in a `typeof` expression (§7.6.10.6) outside of an unsafe context, as such usage is not unsafe.

type:
    ...
    pointer-type

A *pointer-type* is written as an *unmanaged-type* or the keyword `void`, followed by a `*` token:

pointer-type:
    unmanaged-type `*`
    `void` `*`

unmanaged-type:
    type

The type specified before the `*` in a pointer type is called the ***referent type*** of the pointer type. It represents the type of the variable to which a value of the pointer type points.

Unlike references (values of reference types), pointers are not tracked by the garbage collector—the garbage collector has no knowledge of pointers and the data to which they point. For this reason a pointer is not permitted to point to a reference or to a struct that contains references, and the referent type of a pointer must be an *unmanaged-type*.

An *unmanaged-type* is any type that isn't a *reference-type* or constructed type, and doesn't contain *reference-type* or constructed type fields at any level of nesting. In other words, an *unmanaged-type* is one of the following:

- `sbyte`, `byte`, `short`, `ushort`, `int`, `uint`, `long`, `ulong`, `char`, `float`, `double`, `decimal`, or `bool`.
- Any *enum-type*.
- Any *pointer-type*.
- Any user-defined *struct-type* that is not a constructed type and contains fields of *unmanaged-type*s only.

www.it-ebooks.info

The intuitive rule for mixing of pointers and references is that referents of references (objects) are permitted to contain pointers, but referents of pointers are not permitted to contain references.

Some examples of pointer types are given in the table below:

Example	Description
`byte*`	Pointer to `byte`
`char*`	Pointer to `char`
`int**`	Pointer to pointer to `int`
`int*[]`	Single-dimensional array of pointers to `int`
`void*`	Pointer to unknown type

For a given implementation, all pointer types must have the same size and representation.

Unlike C and C++, when multiple pointers are declared in the same declaration, in C# the * is written along with the underlying type only, not as a prefix punctuator on each pointer name. For example:

```
int* pi, pj;      // NOT as int *pi, *pj;
```

The value of a pointer having type `T*` represents the *address* of a variable of type `T`. The pointer indirection operator * (§18.5.1) may be used to access this variable. For example, given a variable `P` of type `int*`, the expression `*P` denotes the `int` variable found at the address contained in `P`.

Like an object reference, a pointer may be `null`. Applying the indirection operator to a `null` pointer results in implementation-defined behavior. A pointer with value `null` is represented by all-bits-zero.

The `void*` type represents a pointer to an unknown type. Because the referent type is unknown, the indirection operator cannot be applied to a pointer of type `void*`, nor can any arithmetic be performed on such a pointer. However, a pointer of type `void*` can be cast to any other pointer type (and vice versa).

www.it-ebooks.info

Pointer types are a separate category of types. Unlike reference types and value types, pointer types do not inherit from `object` and no conversions exist between pointer types and `object`. In particular, boxing and unboxing (§4.3) are not supported for pointers. However, conversions are permitted between different pointer types and between pointer types and the integral types. This is described in §18.4.

A *pointer-type* cannot be used as a type argument (§4.4), and type inference (§7.5.2) fails on generic method calls that would have inferred a type argument to be a pointer type.

A *pointer-type* may be used as the type of a volatile field (§10.5.3).

Although pointers can be passed as `ref` or `out` parameters, doing so can cause undefined behavior, since the pointer may well be set to point to a local variable that no longer exists when the called method returns, or when the fixed object to which it used to point is no longer fixed. For example:

```
using System;

class Test
{
    static int value = 20;

    unsafe static void F(out int* pi1, ref int* pi2)
    {
        int i = 10;
        pi1 = &i;

        fixed (int* pj = &value)
        {
            // ...
            pi2 = pj;
        }
    }

    static void Main()
    {
        int i = 10;
        unsafe
        {
            int* px1;
            int* px2 = &i;

            F(out px1, ref px2);

            Console.WriteLine("*px1 = {0}, *px2 = {1}", *px1, *px2);
            // Undefined behavior
        }
    }
}
```


www.it-ebooks.info

A method can return a value of some type, and that type can be a pointer. For example, when given a pointer to a contiguous sequence of `int`s, that sequence's element count, and some other `int` value, the following method returns the address of that value in that sequence, if a match occurs; otherwise, it returns `null`:

```
unsafe static int* Find(int* pi, int size, int value) {
      for (int i = 0; i < size; ++i) {
            if (*pi == value)
                  return pi;
            ++pi;
      }
      return null;
}
```

In an unsafe context, several constructs are available for operating on pointers:

- The `*` operator may be used to perform pointer indirection (§18.5.1).
- The `->` operator may be used to access a member of a struct through a pointer (§18.5.2).
- The `[]` operator may be used to index a pointer (§18.5.3).
- The `&` operator may be used to obtain the address of a variable (§18.5.4).
- The `++` and `--` operators may be used to increment and decrement pointers (§18.5.5).
- The `+` and `-` operators may be used to perform pointer arithmetic (§18.5.6).
- The `==`, `!=`, `<`, `>`, `<=`, and `=>` operators may be used to compare pointers (§18.5.7).
- The `stackalloc` operator may be used to allocate memory from the call stack (§18.7).
- The `fixed` statement may be used to temporarily fix a variable so its address can be obtained (§18.6).

18.3 Fixed and Moveable Variables

The address-of operator (§18.5.4) and the `fixed` statement (§18.6) divide variables into two categories: ***fixed variables*** and ***moveable variables***.

Fixed variables reside in storage locations that are unaffected by operation of the garbage collector. (Examples of fixed variables include local variables, value parameters, and variables created by dereferencing pointers.) In contrast, moveable variables reside in storage locations that are subject to relocation or disposal by the garbage collector. (Examples of moveable variables include fields in objects and elements of arrays.)

www.it-ebooks.info

The `&` operator (§18.5.4) permits the address of a fixed variable to be obtained without restrictions. However, because a moveable variable is subject to relocation or disposal by the garbage collector, the address of a moveable variable can be obtained only by using a `fixed` statement (§18.6), and that address remains valid only for the duration of that `fixed` statement.

In precise terms, a fixed variable is one of the following:

- A variable resulting from a *simple-name* (§7.6.2) that refers to a local variable or a value parameter, unless the variable is captured by an anonymous function.
- A variable resulting from a *member-access* (§7.6.4) of the form `V.I`, where `V` is a fixed variable of a *struct-type*.
- A variable resulting from a *pointer-indirection-expression* (§18.5.1) of the form `*P`, a *pointer-member-access* (§18.5.2) of the form `P->I`, or a *pointer-element-access* (§18.5.3) of the form `P[E]`.

All other variables are classified as moveable variables.

Note that a static field is classified as a moveable variable. Also note that a `ref` or `out` parameter is classified as a moveable variable, even if the argument given for the parameter is a fixed variable. Finally, note that a variable produced by dereferencing a pointer is always classified as a fixed variable.

18.4 Pointer Conversions

In an unsafe context, the set of available implicit conversions (§6.1) is extended to include the following implicit pointer conversions:

- From any *pointer-type* to the type `void*`.
- From the `null` literal to any *pointer-type*.

Additionally, in an unsafe context, the set of available explicit conversions (§6.2) is extended to include the following explicit pointer conversions:

- From any *pointer-type* to any other *pointer-type*.
- From `sbyte`, `byte`, `short`, `ushort`, `int`, `uint`, `long`, or `ulong` to any *pointer-type*.
- From any *pointer-type* to `sbyte`, `byte`, `short`, `ushort`, `int`, `uint`, `long`, or `ulong`.

www.it-ebooks.info

Finally, in an unsafe context, the set of standard implicit conversions (§6.3.1) includes the following pointer conversion:

- From any *pointer-type* to the type `void*`.

Conversions between two pointer types never change the actual pointer value. In other words, a conversion from one pointer type to another has no effect on the underlying address given by the pointer.

When one pointer type is converted to another, if the resulting pointer is not correctly aligned for the pointed-to type, the behavior is undefined if the result is dereferenced. In general, the concept "correctly aligned" is transitive: If a pointer to type `A` is correctly aligned for a pointer to type `B`, which in turn is correctly aligned for a pointer to type `C`, then a pointer to type `A` is correctly aligned for a pointer to type `C`.

Consider the following case in which a variable having one type is accessed via a pointer to a different type:

```
char c = 'A';
char* pc = &c;
void* pv = pc;
int* pi = (int*)pv;
int i = *pi;            // Undefined
*pi = 123456;           // Undefined
```

When a pointer type is converted to a pointer to byte, the result points to the lowest addressed byte of the variable. Successive increments of the result, up to the size of the variable, yield pointers to the remaining bytes of that variable. For example, the following method displays each of the eight bytes in a double as a hexadecimal value:

```
using System;

class Test
{
    unsafe static void Main()
    {
        double d = 123.456e23;
        unsafe
        {
            byte* pb = (byte*)&d;
            for (int i = 0; i < sizeof(double); ++i)
                Console.Write("{0:X2} ", *pb++);
            Console.WriteLine();
        }
    }
}
```

Of course, the output produced depends on endianness.

www.it-ebooks.info

PETER SESTOFT One use of unsafe code is to perform conversions by reinterpreting bit patterns that represent primitive data, structs, or references, by unsafely type-casting pointers. In most cases, the result will be unportable or meaningless. However, such reinterpretation can be used to portably get and set the NaN payload bits of IEEE floating point numbers (see the annotation in §7.8.1):

```
unsafe static int GetNaNPayload(float f) {
  float* p = &f;
  return *((int*)p) & 0x003FFFFF;
}

unsafe static float MakeNaNPayload(int nanbits) {
  float nan = Single.NaN;
  float* p = &nan;
  *((int*)p) |= (nanbits & 0x003FFFFF);
  return nan;
}
```

Unsafe code does not have to be this obscure, but it *is* likely to be wrong because the standard sanity checks are turned off.

Mappings between pointers and integers are implementation-defined. However, on 32- and 64-bit CPU architectures with a linear address space, conversions of pointers to or from integral types typically behave exactly like conversions of `uint` or `ulong` values, respectively, to or from those integral types.

18.4.1 Pointer Arrays

In an unsafe context, arrays of pointers can be constructed. Only some of the conversions that apply to other array types are allowed on pointer arrays:

- The implicit reference conversion (§6.1.6) from any *array-type* to `System.Array` and the interfaces it implements also applies to pointer arrays. However, any attempt to access the array elements through `System.Array` or the interfaces it implements will result in an exception at runtime, as pointer types are not convertible to `object`.
- The implicit and explicit reference conversions (§6.1.6, §6.2.4) from a single-dimensional array type `S[]` to `System.Collections.Generic.IList<T>` and its base interfaces never apply to pointer arrays, since pointer types cannot be used as type arguments, and there are no conversions from pointer types to non-pointer types.
- The explicit reference conversion (§6.2.4) from `System.Array` and the interfaces it implements to any *array-type* applies to pointer arrays.

www.it-ebooks.info

- The explicit reference conversions (§6.2.4) from `System.Collections.Generic.IList<S>` and its base interfaces to a single-dimensional array type `T[]` never apply to pointer arrays, since pointer types cannot be used as type arguments, and there are no conversions from pointer types to non-pointer types.

These restrictions mean that the expansion for the `foreach` statement over arrays described in §8.8.4 cannot be applied to pointer arrays. Instead, a `foreach` statement of the form

```
foreach (V v in x) embedded-statement
```

where the type of `x` is an array type of the form `T[,,...,]`, *n* is the number of dimensions minus 1, and `T` or `V` is a pointer type, is expanded using nested `for` loops as follows:

```
{
    T[,,...,] a = x;
    V v;
    for (int i0 = a.GetLowerBound(0); i0 <= a.GetUpperBound(0); i0++)
    for (int i1 = a.GetLowerBound(1); i1 <= a.GetUpperBound(1); i1++)
    ...
    for (int in = a.GetLowerBound(n); in <= a.GetUpperBound(n); in++) {
        v  = (V)a.GetValue(i0,i1,...,in);
        embedded-statement
    }
}
```

The variables `a`, `i0`, `i1`, … `in` are not visible to or accessible to `x` or the *embedded-statement* or any other source code of the program. The variable `v` is read-only in the embedded statement. If there is not an explicit conversion (§18.4) from `T` (the element type) to `V`, an error is produced and no further steps are taken. If `x` has the value `null`, a `System.NullReferenceException` is thrown at runtime.

18.5 Pointers in Expressions

In an unsafe context, an expression may yield a result of a pointer type. Outside an unsafe context, it is a compile-time error for an expression to be of a pointer type. In precise terms, outside an unsafe context, a compile-time error occurs if any *simple-name* (§7.6.2), *member-access* (§7.6.4), *invocation-expression* (§7.6.5), or *element-access* (§7.6.6) is of a pointer type.

In an unsafe context, the *primary-no-array-creation-expression* (§7.6) and *unary-expression* (§7.7) productions permit the following additional constructs:

primary-no-array-creation-expression:
    ...
    pointer-member-access
    pointer-element-access
    sizeof-expression

www.it-ebooks.info

unary-expression:
    ...
    pointer-indirection-expression
    addressof-expression

These constructs are described in the following sections. The precedence and associativity of the unsafe operators are implied by the grammar.

18.5.1 Pointer Indirection

A *pointer-indirection-expression* consists of an asterisk (*) followed by a *unary-expression*.

pointer-indirection-expression:
    * *unary-expression*

The unary * operator denotes *pointer indirection* and is used to obtain the variable to which a pointer points. The result of evaluating `*P`, where `P` is an expression of a pointer type `T*`, is a variable of type `T`. It is a compile-time error to apply the unary * operator to an expression of type `void*` or to an expression that isn't of a pointer type.

The effect of applying the unary * operator to a `null` pointer is implementation-defined. In particular, there is no guarantee that this operation throws a `System.NullReferenceException`.

If an invalid value has been assigned to the pointer, the behavior of the unary * operator is undefined. Among the invalid values for dereferencing a pointer by the unary * operator are an address inappropriately aligned for the type pointed to (see example in §18.4), and the address of a variable after the end of its lifetime.

For purposes of definite assignment analysis, a variable produced by evaluating an expression of the form `*P` is considered initially assigned (§5.3.1).

18.5.2 Pointer Member Access

A *pointer-member-access* consists of a *primary-expression*, followed by a "`->`" token, followed by an *identifier*.

pointer-member-access:
    primary-expression `->` *identifier* *type-argument-list*$_{opt}$

In a pointer member access of the form `P->I`, `P` must be an expression of a pointer type other than `void*`, and `I` must denote an accessible member of the type to which `P` points.

> **VLADIMIR RESHETNIKOV** You cannot apply *pointer-member-access* to pointer-to-pointer types (like `int**`), because pointers have no members.

www.it-ebooks.info

A pointer member access of the form `P->I` is evaluated exactly as `(*P).I`. For a description of the pointer indirection operator (*), see §18.5.1. For a description of the member access operator (.), see §7.6.4.

In the example

```
using System;

struct Point
{
    public int x;
    public int y;

    public override string ToString()
    {
        return "(" + x + "," + y + ")";
    }
}

class Test
{
    static void Main()
    {
        Point point;
        unsafe
        {
            Point* p = &point;
            p->x = 10;
            p->y = 20;
            Console.WriteLine(p->ToString());
        }
    }
}
```

the -> operator is used to access fields and invoke a method of a struct through a pointer. Because the operation `P->I` is precisely equivalent to `(*P).I`, the `Main` method could equally well have been written like this:

```
class Test
{
    static void Main()
    {
        Point point;
        unsafe
        {
            Point* p = &point;
            (*p).x = 10;
            (*p).y = 20;
            Console.WriteLine((*p).ToString());
        }
    }
}
```


www.it-ebooks.info

18.5.3 Pointer Element Access

A *pointer-element-access* consists of a *primary-no-array-creation-expression* followed by an expression enclosed in "[" and "]".

pointer-element-access:
primary-no-array-creation-expression [*expression*]

In a pointer element access of the form `P[E]`, `P` must be an expression of a pointer type other than `void*`, and `E` must be an expression that can be implicitly converted to `int`, `uint`, `long`, or `ulong`.

A pointer element access of the form `P[E]` is evaluated exactly as `*(P + E)`. For a description of the pointer indirection operator (*), see §18.5.1. For a description of the pointer addition operator (+), see §18.5.6.

In the example

```
class Test
{
    static void Main()
    {
        unsafe
        {
            char* p = stackalloc char[256];
            for (int i = 0; i < 256; i++) p[i] = (char)i;
        }
    }
}
```

a pointer element access is used to initialize the character buffer in a `for` loop. Because the operation `P[E]` is precisely equivalent to `*(P + E)`, the example could equally well have been written like this:

```
class Test
{
    static void Main()
    {
        unsafe
        {
            char* p = stackalloc char[256];
            for (int i = 0; i < 256; i++) *(p + i) = (char)i;
        }
    }
}
```

The pointer element access operator does not check for out-of-bounds errors, and the behavior when accessing an out-of-bounds element is undefined. This is the same approach as used in C and C++.

www.it-ebooks.info

18.5.4 The Address-of Operator

An *addressof-expression* consists of an ampersand (`&`) followed by a *unary-expression*.

addressof-expression:
`&` *unary-expression*

Given an expression `E` that is of a type `T` and is classified as a fixed variable (§18.3), the construct `&E` computes the address of the variable given by `E`. The type of the result is `T*` and is classified as a value. A compile-time error occurs if `E` is not classified as a variable, if `E` is classified as a read-only local variable, or if `E` denotes a moveable variable. In the last case, a fixed statement (§18.6) can be used to temporarily "fix" the variable before obtaining its address. As stated in §7.6.4, outside an instance constructor or static constructor for a struct or class that defines a `readonly` field, that field is considered a value, not a variable. As such, its address cannot be taken. Similarly, the address of a constant cannot be taken.

The `&` operator does not require its argument to be definitely assigned, but following an `&` operation, the variable to which the operator is applied is considered definitely assigned in the execution path in which the operation occurs. It is the responsibility of the programmer to ensure that correct initialization of the variable actually does take place in this situation.

In the example

```
using System;

class Test
{
    static void Main()
    {
        int i;
        unsafe
        {
            int* p = &i;
            *p = 123;
        }
        Console.WriteLine(i);
    }
}
```

`i` is considered definitely assigned following the `&i` operation used to initialize `p`. The assignment to `*p` in effect initializes `i`, but the inclusion of this initialization is the responsibility of the programmer, and no compile-time error would occur if the assignment was removed.

The rules of definite assignment for the `&` operator exist such that redundant initialization of local variables can be avoided. For example, many external APIs take a pointer to a structure that is filled in by the API. Calls to such APIs typically pass the address of a local

www.it-ebooks.info

struct variable, and without the rule, redundant initialization of the struct variable would be required.

18.5.5 Pointer Increment and Decrement

In an unsafe context, the `++` and `--` operators (§7.6.9 and §7.7.5) can be applied to pointer variables of all types except `void*`. Thus, for every pointer type `T*`, the following operators are implicitly defined:

```
T* operator ++(T* x);
T* operator --(T* x);
```

The operators produce the same results as `x + 1` and `x - 1`, respectively (§18.5.6). In other words, for a pointer variable of type `T*`, the `++` operator adds `sizeof(T)` to the address contained in the variable, and the `--` operator subtracts `sizeof(T)` from the address contained in the variable.

If a pointer increment or decrement operation overflows the domain of the pointer type, the result is implementation-defined, but no exceptions are produced.

18.5.6 Pointer Arithmetic

In an unsafe context, the `+` and `-` operators (§7.8.4 and §7.8.5) can be applied to values of all pointer types except `void*`. Thus, for every pointer type `T*`, the following operators are implicitly defined:

```
T* operator +(T* x, int y);
T* operator +(T* x, uint y);
T* operator +(T* x, long y);
T* operator +(T* x, ulong y);

T* operator +(int x, T* y);
T* operator +(uint x, T* y);
T* operator +(long x, T* y);
T* operator +(ulong x, T* y);

T* operator –(T* x, int y);
T* operator –(T* x, uint y);
T* operator –(T* x, long y);
T* operator –(T* x, ulong y);

long operator –(T* x, T* y);
```

Given an expression `P` of a pointer type `T*` and an expression `N` of type `int`, `uint`, `long`, or `ulong`, the expressions `P + N` and `N + P` compute the pointer value of type `T*` that results from adding `N * sizeof(T)` to the address given by `P`. Likewise, the expression `P - N` computes the pointer value of type `T*` that results from subtracting `N * sizeof(T)` from the address given by `P`.

www.it-ebooks.info

Given two expressions, `P` and `Q`, of a pointer type `T*`, the expression `P – Q` computes the difference between the addresses given by `P` and `Q` and then divides that difference by `sizeof(T)`. The type of the result is always `long`. In effect, `P - Q` is computed as `((long)(P) - (long)(Q)) / sizeof(T)`.

For example,

```
using System;

class Test
{
    static void Main()
    {
        unsafe
        {
            int* values = stackalloc int[20];
            int* p = &values[1];
            int* q = &values[15];
            Console.WriteLine("p - q = {0}", p - q);
            Console.WriteLine("q - p = {0}", q - p);
        }
    }
}
```

produces the following output:

```
p - q = -14
q - p = 14
```

If a pointer arithmetic operation overflows the domain of the pointer type, the result is truncated in an implementation-defined fashion, but no exceptions are produced.

18.5.7 Pointer Comparison

In an unsafe context, the `==`, `!=`, `<`, `>`, `<=`, and `=>` operators (§7.10) can be applied to values of all pointer types. The pointer comparison operators are:

```
bool operator ==(void* x, void* y);
bool operator !=(void* x, void* y);
bool operator <(void* x, void* y);
bool operator >(void* x, void* y);
bool operator <=(void* x, void* y);
bool operator >=(void* x, void* y);
```

Because an implicit conversion exists from any pointer type to the `void*` type, operands of any pointer type can be compared using these operators. The comparison operators compare the addresses given by the two operands as if they were unsigned integers.

www.it-ebooks.info

18.5.8 The sizeof Operator

The `sizeof` operator returns the number of bytes occupied by a variable of a given type. The type specified as an operand to `sizeof` must be an *unmanaged-type* (§18.2).

sizeof-expression:
    `sizeof` `(` *unmanaged-type* `)`

The result of the `sizeof` operator is a value of type `int`. For certain predefined types, the `sizeof` operator yields a constant value as shown in the table below.

Expression	Result
`sizeof(sbyte)`	`1`
`sizeof(byte)`	`1`
`sizeof(short)`	`2`
`sizeof(ushort)`	`2`
`sizeof(int)`	`4`
`sizeof(uint)`	`4`
`sizeof(long)`	`8`
`sizeof(ulong)`	`8`
`sizeof(char)`	`2`
`sizeof(float)`	`4`
`sizeof(double)`	`8`
`sizeof(bool)`	`1`

For all other types, the result of the `sizeof` operator is implementation-defined and is classified as a value, not a constant.

The order in which members are packed into a struct is unspecified.

For alignment purposes, there may be unnamed padding at the beginning of a struct, within a struct, and at the end of the struct. The contents of the bits used as padding are indeterminate.

When applied to an operand that has struct type, the result is the total number of bytes in a variable of that type, including any padding.

www.it-ebooks.info

18.6 The fixed Statement

In an unsafe context, the *embedded-statement* (§8) production permits an additional construct, the `fixed` statement, which is used to "fix" a moveable variable such that its address remains constant for the duration of the statement.

embedded-statement:
    ...
    fixed-statement

fixed-statement:
    `fixed` `(` *pointer-type* *fixed-pointer-declarators* `)` *embedded-statement*

fixed-pointer-declarators:
    fixed-pointer-declarator
    fixed-pointer-declarators `,` *fixed-pointer-declarator*

fixed-pointer-declarator:
    identifier `=` *fixed-pointer-initializer*

fixed-pointer-initializer:
    `&` *variable-reference*
    expression

Each *fixed-pointer-declarator* declares a local variable of the given *pointer-type* and initializes that local variable with the address computed by the corresponding *fixed-pointer-initializer*. A local variable declared in a `fixed` statement is accessible in any *fixed-pointer-initializer*s occurring to the right of that variable's declaration, and in the *embedded-statement* of the `fixed` statement. A local variable declared by a `fixed` statement is considered read-only. A compile-time error occurs if the embedded statement attempts to modify this local variable (via assignment or the `++` and `--` operators) or pass it as a `ref` or `out` parameter.

A *fixed-pointer-initializer* can be one of the following:

- The token "`&`" followed by a *variable-reference* (§5.3.3) to a moveable variable (§18.3) of an unmanaged type `T`, provided the type `T*` is implicitly convertible to the pointer type given in the `fixed` statement. In this case, the initializer computes the address of the given variable, and the variable is guaranteed to remain at a fixed address for the duration of the `fixed` statement.
- An expression of an *array-type* with elements of an unmanaged type `T`, provided the type `T*` is implicitly convertible to the pointer type given in the `fixed` statement. In this case, the initializer computes the address of the first element in the array, and the entire array is guaranteed to remain at a fixed address for the duration of the `fixed` statement. The

www.it-ebooks.info

behavior of the `fixed` statement is implementation-defined if the array expression is null or if the array has zero elements.

- An expression of type `string`, provided the type `char*` is implicitly convertible to the pointer type given in the `fixed` statement. In this case, the initializer computes the address of the first character in the string, and the entire string is guaranteed to remain at a fixed address for the duration of the `fixed` statement. The behavior of the `fixed` statement is implementation-defined if the string expression is null.
- A *simple-name* or *member-access* that references a fixed size buffer member of a moveable variable, provided the type of the fixed size buffer member is implicitly convertible to the pointer type given in the `fixed` statement. In this case, the initializer computes a pointer to the first element of the fixed size buffer (§18.7.2), and the fixed size buffer is guaranteed to remain at a fixed address for the duration of the `fixed` statement.

ERIC LIPPERT This point demonstrates a potentially confusing conflation: The keyword `fixed` is used to mean both "fixed in location" and "fixed in size." Adding to the confusion is the fact that a fixed-in-size array member must be fixed-in-location to be used. Try to keep in mind that a fixed-in-size block is not automatically fixed-in-place.

For each address computed by a *fixed-pointer-initializer,* the `fixed` statement ensures that the variable referenced by the address is not subject to relocation or disposal by the garbage collector for the duration of the `fixed` statement. For example, if the address computed by a *fixed-pointer-initializer* references a field of an object or an element of an array instance, the `fixed` statement guarantees that the containing object instance will not be relocated or disposed of during the lifetime of the statement.

It is the programmer's responsibility to ensure that pointers created by `fixed` statements do not survive beyond execution of those statements. For example, when pointers created by `fixed` statements are passed to external APIs, it is the programmer's responsibility to ensure that the APIs retain no memory of these pointers.

ERIC LIPPERT If you are in the unfortunate position of having to keep a block of managed memory fixed in place for longer than the duration of a `fixed` statement, then you can do so via the `GCHandle` type. That said, it is best to avoid getting into that situation in the first place.

Fixed objects may cause fragmentation of the heap (because they cannot be moved). For that reason, objects should be fixed only when absolutely necessary, and then only for the shortest amount of time possible.

www.it-ebooks.info

The example

```
class Test
{
    static int x;
    int y;

    unsafe static void F(int* p)
    {
        *p = 1;
    }

    static void Main()
    {
        Test t = new Test();
        int[] a = new int[10];
        unsafe
        {
            fixed (int* p = &x) F(p);
            fixed (int* p = &t.y) F(p);
            fixed (int* p = &a[0]) F(p);
            fixed (int* p = a) F(p);
        }
    }
}
```

demonstrates several uses of the `fixed` statement. The first statement fixes and obtains the address of a static field, the second statement fixes and obtains the address of an instance field, and the third statement fixes and obtains the address of an array element. In each case it would have been an error to use the regular `&` operator since the variables are all classified as moveable variables.

The fourth `fixed` statement in the example above produces a similar result to the third.

This example of the `fixed` statement uses `string`:

```
class Test
{
    static string name = "xx";

    unsafe static void F(char* p)
    {
        for (int i = 0; p[i] != '\0'; ++i)
            Console.WriteLine(p[i]);
    }

    static void Main()
    {
        unsafe
        {
            fixed (char* p = name) F(p);
            fixed (char* p = "xx") F(p);
        }
    }
}
```


www.it-ebooks.info

In an unsafe context, array elements of single-dimensional arrays are stored in increasing index order, starting with index `0` and ending with index `Length – 1`. For multi-dimensional arrays, array elements are stored such that the indices of the rightmost dimension are increased first, then the next left dimension, and so on to the left. Within a `fixed` statement that obtains a pointer `p` to an array instance `a`, the pointer values ranging from `p` to `p + a.Length - 1` represent addresses of the elements in the array. Likewise, the variables ranging from `p[0]` to `p[a.Length - 1]` represent the actual array elements. Given the way in which arrays are stored, we can treat an array of any dimension as though it were linear.

For example,

```
using System;

class Test
{
    static void Main()
    {
        int[, ,] a = new int[2, 3, 4];
        unsafe
        {
            fixed (int* p = a)
            {
                for (int i = 0; i < a.Length; ++i)   // Treat as linear
                    p[i] = i;
            }
        }

        for (int i = 0; i < 2; ++i)
            for (int j = 0; j < 3; ++j)
            {
                for (int k = 0; k < 4; ++k)
                    Console.Write("[{0},{1},{2}] = {3,2} ", i, j, k, a[i, j, k]);
                Console.WriteLine();
            }
    }
}
```

produces the following output:

```
[0,0,0] =  0 [0,0,1] =  1 [0,0,2] =  2 [0,0,3] =  3
[0,1,0] =  4 [0,1,1] =  5 [0,1,2] =  6 [0,1,3] =  7
[0,2,0] =  8 [0,2,1] =  9 [0,2,2] = 10 [0,2,3] = 11
[1,0,0] = 12 [1,0,1] = 13 [1,0,2] = 14 [1,0,3] = 15
[1,1,0] = 16 [1,1,1] = 17 [1,1,2] = 18 [1,1,3] = 19
[1,2,0] = 20 [1,2,1] = 21 [1,2,2] = 22 [1,2,3] = 23
```


www.it-ebooks.info

In the example

```
class Test
{
    unsafe static void Fill(int* p, int count, int value)
    {
        for (; count != 0; count--) *p++ = value;
    }

    static void Main()
    {
        int[] a = new int[100];
        unsafe
        {
            fixed (int* p = a) Fill(p, 100, -1);
        }
    }
}
```

a `fixed` statement is used to fix an array so its address can be passed to a method that takes a pointer.

In the example

```
unsafe struct Font
{
    public int size;
    public fixed char name[32];
}

class Test
{
    unsafe static void PutString(string s, char* buffer, int bufSize)
    {
        int len = s.Length;
        if (len > bufSize) len = bufSize;
        for (int i = 0; i < len; i++) buffer[i] = s[i];
        for (int i = len; i < bufSize; i++) buffer[i] = (char)0;
    }

    Font f;

    unsafe static void Main()
    {
        Test test = new Test();
        test.f.size = 10;
        fixed (char* p = test.f.name)
        {
            PutString("Times New Roman", p, 32);
        }
    }
}
```


www.it-ebooks.info

a `fixed` statement is used to fix a fixed-size buffer of a struct so that its address can be used as a pointer.

A `char*` value produced by fixing a string instance always points to a null-terminated string. Within a `fixed` statement that obtains a pointer `p` to a string instance `s`, the pointer values ranging from `p` to `p + s.Length - 1` represent addresses of the characters in the string, and the pointer value `p + s.Length` always points to a null character (the character with value `'\0'`).

Modifying objects of managed type through fixed pointers can result in undefined behavior. For example, because strings are immutable, it is the programmer's responsibility to ensure that the characters referenced by a pointer to a fixed string are not modified.

The automatic null-termination of strings is particularly convenient when calling external APIs that expect "C-style" strings. Note, however, that a string instance is permitted to contain null characters. If such null characters are present, the string will appear truncated when treated as a null-terminated `char*`.

18.7 Fixed-Size Buffers

Fixed-size buffers are used to declare "C style" in-line arrays as members of structs, and are primarily useful for interfacing with unmanaged APIs.

18.7.1 Fixed-Size Buffer Declarations

A ***fixed-size buffer*** is a member that represents storage for a fixed-length buffer of variables of a given type. A fixed-size buffer declaration introduces one or more fixed-size buffers of a given element type. Fixed-size buffers are permitted only in struct declarations and can occur only in unsafe contexts (§18.1).

struct-member-declaration:
    …
    fixed-size-buffer-declaration

fixed-size-buffer-declaration:
    attributes$_{opt}$ *fixed-size-buffer-modifiers*$_{opt}$ `fixed` *buffer-element-type*
        fixed-size-buffer-declarators `;`

fixed-size-buffer-modifiers:
    fixed-size-buffer-modifier
    fixed-size-buffer-modifier *fixed-size-buffer-modifiers*

www.it-ebooks.info

fixed-size-buffer-modifier:
    `new`
    `public`
    `protected`
    `internal`
    `private`
    `unsafe`

buffer-element-type:
    type

fixed-size-buffer-declarators:
    fixed-size-buffer-declarator
    fixed-size-buffer-declarator `,` *fixed-size-buffer-declarators*

fixed-size-buffer-declarator:
    identifier `[` *constant-expression* `]`

A fixed-size buffer declaration may include a set of attributes (§17), a `new` modifier (§10.2.2), a valid combination of the four access modifiers (§10.2.3), and an `unsafe` modifier (§18.1). The attributes and modifiers apply to all of the members declared by the fixed-size buffer declaration. It is an error for the same modifier to appear multiple times in a fixed-size buffer declaration.

A fixed-size buffer declaration is not permitted to include the `static` modifier.

The buffer element type of a fixed-size buffer declaration specifies the element type of the buffer(s) introduced by the declaration. The buffer element type must be one of the predefined types `sbyte`, `byte`, `short`, `ushort`, `int`, `uint`, `long`, `ulong`, `char`, `float`, `double`, or `bool`.

The buffer element type is followed by a list of fixed-size buffer declarators, each of which introduces a new member. A fixed-size buffer declarator consists of an identifier that names the member, followed by a constant expression enclosed in "`[`" and "`]`" tokens. The constant expression denotes the number of elements in the member introduced by that fixed-size buffer declarator. The type of the constant expression must be implicitly convertible to type `int`, and the value must be a non-zero positive integer.

The elements of a fixed-size buffer are guaranteed to be laid out sequentially in memory.

A fixed-size buffer declaration that declares multiple fixed-size buffers is equivalent to multiple declarations of a single fixed-size buffer with the same attributes and element types. For example,

www.it-ebooks.info

```
unsafe struct A
{
    public fixed int x[5], y[10], z[100];
}
```

is equivalent to

```
unsafe struct A
{
    public fixed int x[5];
    public fixed int y[10];
    public fixed int z[100];
}
```

18.7.2 Fixed-Size Buffers in Expressions

Member lookup (§7.3) of a fixed-size buffer member proceeds exactly like member lookup of a field.

A fixed-size buffer can be referenced in an expression using a *simple-name* (§7.5.2) or a *member-access* (§7.5.4).

When a fixed-size buffer member is referenced as a simple name, the effect is the same as a member access of the form `this.I`, where `I` is the fixed-size buffer member.

In a member access of the form `E.I`, if `E` is of a struct type and a member lookup of `I` in that struct type identifies a fixed-size member, then `E.I` is evaluated and classified as follows:

- If the expression `E.I` does not occur in an unsafe context, a compile-time error occurs.
- If `E` is classified as a value, a compile-time error occurs.
- Otherwise, if `E` is a moveable variable (§18.3) and the expression `E.I` is not a *fixed-pointer-initializer* (§18.6), a compile-time error occurs.
- Otherwise, `E` references a fixed variable and the result of the expression is a pointer to the first element of the fixed-size buffer member `I` in `E`. The result is of type `S*`, where `S` is the element type of `I`, and is classified as a value.

The subsequent elements of the fixed-size buffer can be accessed using pointer operations from the first element. Unlike access to arrays, access to the elements of a fixed-size buffer is an unsafe operation and is not range checked.

The following example declares and uses a struct with a fixed-size buffer member:

```
unsafe struct Font
{
    public int size;
    public fixed char name[32];
}
```


www.it-ebooks.info

```
class Test
{
    unsafe static void PutString(string s, char* buffer, int bufSize)
    {
        int len = s.Length;
        if (len > bufSize) len = bufSize;
        for (int i = 0; i < len; i++) buffer[i] = s[i];
        for (int i = len; i < bufSize; i++) buffer[i] = (char)0;
    }

    unsafe static void Main()
    {
        Font f;
        f.size = 10;
        PutString("Times New Roman", f.name, 32);
    }
}
```


18.7.3 Definite Assignment Checking

Fixed-size buffers are not subject to definite assignment checking (§5.3), and fixed-size buffer members are ignored for purposes of definite assignment checking of struct type variables.

When the outermost containing struct variable of a fixed-size buffer member is a static variable, an instance variable of a class instance, or an array element, the elements of the fixed-size buffer are automatically initialized to their default values (§5.2). In all other cases, the initial content of a fixed-size buffer is undefined.

18.8 Stack Allocation

In an unsafe context, a local variable declaration (§8.5.1) may include a stack allocation initializer that allocates memory from the call stack.

ERIC LIPPERT This is a bit overspecific; there is no requirement that a particular CPU allow allocation of arbitrary memory on its *call stack.* For example, the CLR could be implemented on a CPU architecture that keeps *two* stacks—one for storage of local variables and one for tracking return addresses. (Such architectures prevent the stack-rewriting attacks that plagued the x86 architecture.) Really, what we ought to say here is simply that an immobile *local memory store* is associated with each method invocation, and that `stackalloc` allocates memory out of that store. In a typical implementation, that store will be the call stack, but that's an implementation detail.

www.it-ebooks.info

local-variable-initializer:
    …
    stackalloc-initializer

stackalloc-initializer:
    `stackalloc` *unmanaged-type* `[` *expression* `]`

The *unmanaged-type* indicates the type of the items that will be stored in the newly allocated location, and the *expression* indicates the number of these items. Taken together, these specify the required allocation size. Since the size of a stack allocation cannot be negative, it is a compile-time error to specify the number of items as a *constant-expression* that evaluates to a negative value.

A stack allocation initializer of the form `stackalloc T[E]` requires `T` to be an unmanaged type (§18.2) and `E` to be an expression of type `int`. The construct allocates `E * sizeof(T)` bytes from the call stack and returns a pointer, of type `T*`, to the newly allocated block. If `E` is a negative value, then the behavior is undefined. If `E` is zero, then no allocation is made, and the pointer returned is implementation-defined. If there is not enough memory available to allocate a block of the given size, a `System.StackOverflowException` is thrown.

The content of the newly allocated memory is undefined.

Stack allocation initializers are not permitted in `catch` or `finally` blocks (§8.10).

There is no way to explicitly free memory allocated using `stackalloc`. All stack allocated memory blocks created during the execution of a function member are automatically discarded when that function member returns. This corresponds to the `alloca` function, an extension commonly found in C and C++ implementations.

In the example

```
using System;

class Test
{
    static string IntToString(int value)
    {
        int n = value >= 0 ? value : -value;
        unsafe
        {
            char* buffer = stackalloc char[16];
            char* p = buffer + 16;
            do
            {
                *--p = (char)(n % 10 + '0');
                n /= 10;
            } while (n != 0);
```


www.it-ebooks.info

```
            if (value < 0) *--p = '-';
            return new string(p, 0, (int)(buffer + 16 - p));
        }
    }

    static void Main()
    {
        Console.WriteLine(IntToString(12345));
        Console.WriteLine(IntToString(-999));
    }
}
```

a `stackalloc` initializer is used in the `IntToString` method to allocate a buffer of 16 characters on the stack. The buffer is automatically discarded when the method returns.

18.9 Dynamic Memory Allocation

Except for the `stackalloc` operator, C# provides no predefined constructs for managing non-garbage-collected memory. Such services are typically provided by supporting class libraries or imported directly from the underlying operating system. For example, the `Memory` class below illustrates how the heap functions of an underlying operating system might be accessed from C#:

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

public unsafe class Memory
{
    // Handle for the process heap. This handle is used
    // in all calls to the
    // HeapXXX APIs in the methods below.

    static int ph = GetProcessHeap();

    // Private instance constructor to prevent instantiation.
    private Memory() { }

    // Allocates a memory block of the given size.
    // The allocated memory is
    // automatically initialized to zero.

    public static void* Alloc(int size)
    {
        void* result = HeapAlloc(ph, HEAP_ZERO_MEMORY, size);
        if (result == null) throw new OutOfMemoryException();
        return result;
    }

    // Copies count bytes from src to dst. The source and destination
    // blocks are permitted to overlap.
```


www.it-ebooks.info

```
public static void Copy(void* src, void* dst, int count)
{
    byte* ps = (byte*)src;
    byte* pd = (byte*)dst;
    if (ps > pd)
    {
        for (; count != 0; count--) *pd++ = *ps++;
    }
    else if (ps < pd)
    {
        for (ps += count, pd += count; count != 0; count--) *--pd = *--ps;
    }
}

// Frees a memory block.

public static void Free(void* block)
{
    if (!HeapFree(ph, 0, block)) throw new InvalidOperationException();
}

// Reallocates a memory block. If the reallocation
// request is for a larger size, the additional region of
// memory is automatically initialized to zero.

public static void* ReAlloc(void* block, int size)
{
    void* result = HeapReAlloc(ph, HEAP_ZERO_MEMORY, block, size);
    if (result == null) throw new OutOfMemoryException();
    return result;
}

// Returns the size of a memory block.

public static int SizeOf(void* block)
{
    int result = HeapSize(ph, 0, block);
    if (result == -1) throw new InvalidOperationException();
    return result;
}

// Heap API flags

const int HEAP_ZERO_MEMORY = 0x00000008;

// Heap API functions

[DllImport("kernel32")]
static extern int GetProcessHeap();

[DllImport("kernel32")]
static extern void* HeapAlloc(int hHeap, int flags, int size);

[DllImport("kernel32")]
static extern bool HeapFree(int hHeap, int flags, void* block);
```


www.it-ebooks.info

```
    [DllImport("kernel32")]
    static extern void* HeapReAlloc(int hHeap, int flags,
        void* block, int size);

    [DllImport("kernel32")]
    static extern int HeapSize(int hHeap, int flags, void* block);
}
```

An example that uses the `Memory` class is given below:

```
class Test
{
    static void Main()
    {
        unsafe
        {
            byte* buffer = (byte*)Memory.Alloc(256);
            try
            {
                for (int i = 0; i < 256; i++) buffer[i] = (byte)i;
                byte[] array = new byte[256];
                fixed (byte* p = array) Memory.Copy(buffer, p, 256);
            }
            finally
            {
                Memory.Free(buffer);
            }
            for (int i = 0; i < 256; i++) Console.WriteLine(array[i]);
        }
    }
}
```

This example allocates 256 bytes of memory through `Memory.Alloc` and initializes the memory block with values increasing from 0 to 255. It then allocates a 256-element byte array and uses `Memory.Copy` to copy the contents of the memory block into the byte array. Finally, the memory block is freed using `Memory.Free` and the contents of the byte array are output on the console.

www.it-ebooks.info

A. Documentation Comments

C# provides a mechanism for programmers to document their code using a special comment syntax that contains XML text. In source code files, comments having a certain form can be used to direct a tool to produce XML from those comments and the source code elements, which they precede. Comments using such syntax are called ***documentation comments***. They must immediately precede a user-defined type (such as a class, delegate, or interface) or a member (such as a field, event, property, or method). The XML generation tool is called the ***documentation generator***. (This generator could be, but need not be, the C# compiler itself.) The output produced by the documentation generator is called the ***documentation file***. A documentation file is used as input to a ***documentation viewer***—a tool intended to produce some sort of visual display of type information and its associated documentation.

This specification suggests a set of tags to be used in documentation comments. Use of these tags is not required, and other tags may be used if desired, as long the rules of well-formed XML are followed.

A.1 Introduction

Comments having a special form can be used to direct a tool to produce XML from those comments and the source code elements, which they precede. Such comments are single-line comments that start with three slashes (`///`), or delimited comments that start with a slash and two stars (`/**`). They must immediately precede a user-defined type (such as a class, delegate, or interface) or a member (such as a field, event, property, or method) that they annotate. Attribute sections (§17.2) are considered part of declarations, so documentation comments must precede attributes applied to a type or member.

Syntax:

single-line-doc-comment:
    `///` *input-characters*$_{opt}$

delimited-doc-comment:
    `/**` *delimited-comment-text*$_{opt}$ `*/`

www.it-ebooks.info

In a *single-line-doc-comment*, if there is a *whitespace* character following the `///` characters on each of the *single-line-doc-comment*s adjacent to the current *single-line-doc-comment*, then that *whitespace* character is not included in the XML output.

In a *delimited-doc-comment*, if the first non-*whitespace* character on the second line is an *asterisk* and the same pattern of optional *whitespace* characters and an *asterisk* character is repeated at the beginning of each of the lines within the *delimited-doc-comment*, then the characters of the repeated pattern are not included in the XML output. The pattern may include *whitespace* characters after, as well as before, the *asterisk* character.

Example:

```
/// <summary>Class <c>Point</c> models a point in a two-dimensional
/// plane.</summary>
///
public class Point
{
    /// <summary>method <c>draw</c> renders the point.</summary>
    void draw() {…}
}
```

The text within documentation comments must be well formed according to the rules of XML (http://www.w3.org/TR/REC-xml). If the XML is ill formed, a warning is generated and the documentation file will contain a comment saying that an error was encountered.

Although developers are free to create their own set of tags, a recommended set is defined in §A.2. Some of the recommended tags have special meanings:

- The `<param>` tag is used to describe parameters. If such a tag is used, the documentation generator must verify that the specified parameter exists and that all parameters are described in documentation comments. If such verification fails, the documentation generator issues a warning.
- The `cref` attribute can be attached to any tag to provide a reference to a code element. The documentation generator must verify that this code element exists. If the verification fails, the documentation generator issues a warning. When looking for a name described in a `cref` attribute, the documentation generator must respect namespace visibility according to `using` statements appearing within the source code. For code elements that are generic, the normal generic syntax (i.e., "`List<T>`") cannot be used because it produces invalid XML. Braces can be used instead of brackets (i.e., "`List{T}`"), or the XML escape syntax can be used (i.e., "`List<T>`").
- The `<summary>` tag is intended to be used by a documentation viewer to display additional information about a type or member.
- The `<include>` tag includes information from an external XML file.

www.it-ebooks.info

Note that the documentation file does not provide full information about the type and members (for example, it does not contain any type information). To get such information about a type or member, the documentation file must be used in conjunction with reflection on the actual type or member.

A.2 Recommended Tags

The documentation generator must accept and process any tag that is valid according to the rules of XML. The following tags provide commonly used functionality in user documentation. (Of course, other tags are possible.)

Tag	Section	Purpose
`<c>`	A.2.1	Set text in a code-like font
`<code>`	A.2.2	Set one or more lines of source code or program output
`<example>`	A.2.3	Indicate an example
`<exception>`	A.2.4	Identifies the exceptions a method can throw
`<include>`	A.2.5	Includes XML from an external file
`<list>`	A.2.6	Create a list or table
`<para>`	A.2.7	Permit structure to be added to text
`<param>`	A.2.8	Describe a parameter for a method or constructor
`<paramref>`	A.2.9	Identify that a word is a parameter name
`<permission>`	A.2.10	Document the security accessibility of a member
`<remark>`	A.2.11	Describe additional information about a type
`<returns>`	A.2.12	Describe the return value of a method
`<see>`	A.2.13	Specify a link
`<seealso>`	A.2.14	Generate a *See Also* entry
`<summary>`	A.2.15	Describe a type or a member of a type
`<value>`	A.2.16	Describe a property

Continued

www.it-ebooks.info

Tag	Section	Purpose
<typeparam>	A.2.17	Describe a generic type parameter
<typeparamref>	A.2.18	Identify that a word is a type parameter name

A.2.1 <c>

This tag provides a mechanism to indicate that a fragment of text within a description should be set in a special font such as that used for a block of code. For lines of actual code, use <code> (§A.2.2).

Syntax:

```
<c>text</c>
```

Example:

```
/// <summary>Class <c>Point</c> models a point in a two-dimensional
/// plane.</summary>
public class Point
{
    // ...
}
```

A.2.2 <code>

This tag is used to set one or more lines of source code or program output in some special font. For small code fragments in narrative, use <c> (§A.2.1).

Syntax:

```
<code>source code or program output</code>
```

Example:

```
/// <summary>This method changes the point's location by
///   the given x- and y-offsets.
/// <example>For example:
/// <code>
///   Point p = new Point(3,5);
///   p.Translate(-1,3);
/// </code>
/// results in <c>p</c>'s having the value (2,8).
/// </example>
/// </summary>

public void Translate(int xor, int yor) {
    X += xor;
    Y += yor;
}
```


www.it-ebooks.info

A.2.3 <example>

This tag allows example code within a comment to specify how a method or other library member may be used. Ordinarily, this would also involve use of the tag <code> (§A.2.2).

Syntax:

```
<example>description</example>
```

Example:

See <code> (§A.2.2) for an example.

A.2.4 <exception>

This tag provides a way to document the exceptions a method can throw.

Syntax:

```
<exception cref="member">description</exception>
```

where

`cref="member"`

The name of a member. The documentation generator checks that the given member exists and translates *member* to the canonical element name in the documentation file.

description

A description of the circumstances in which the exception is thrown.

Example:

```
public class DataBaseOperations
{
    /// <exception cref="MasterFileFormatCorruptException"></exception>
    /// <exception cref="MasterFileLockedOpenException"></exception>
    public static void ReadRecord(int flag) {
        if (flag == 1)
            throw new MasterFileFormatCorruptException();
        else if (flag == 2)
            throw new MasterFileLockedOpenException();
        // …
    }
}
```

A.2.5 <include>

This tag allows including information from an XML document that is external to the source code file. The external file must be a well-formed XML document, and an XPath expression is applied to that document to specify which XML from that document to include. The <include> tag is then replaced with the selected XML from the external document.

www.it-ebooks.info

Syntax:

```
<include file="filename" path="xpath" />
```

where

`file="filename"`

The file name of an external XML file. The file name is interpreted relative to the file that contains the include tag.

`path="xpath"`

An XPath expression that selects some of the XML in the external XML file.

Example:

If the source code contained a declaration like this:

```
/// <include file="docs.xml" path='extradoc/class[@name="IntList"]/*' />
public class IntList { … }
```

and the external file "docs.xml" had the following contents:

```
<?xml version="1.0"?>
<extradoc>
  <class name="IntList">
    <summary>
      Contains a list of integers.
    </summary>
  </class>
  <class name="StringList">
    <summary>
      Contains a list of integers.
    </summary>
  </class>
</extradoc>
```

then the same documentation is output as if the source code contained:

```
/// <summary>
///    Contains a list of integers.
/// </summary>
public class IntList { … }
```

A.2.6 <list>

This tag is used to create a list or table of items. It may contain a `<listheader>` block to define the heading row of either a table or definition list. (When defining a table, only an entry for *term* in the heading need be supplied.)

www.it-ebooks.info

Each item in the list is specified with an `<item>` block. When creating a definition list, both `term` and *`description`* must be specified. However, for a table, bulleted list, or numbered list, only *`description`* need be specified.

Syntax:

```
<list type="bullet" | "number" | "table">
  <listheader>
    <term>term</term>
    <description>description</description>
  </listheader>
  <item>
    <term>term</term>
    <description>description</description>
  </item>
  …
  <item>
    <term>term</term>
    <description>description</description>
  </item>
</list>
```

where

`term`

The term to define, whose definition is in *`description`*.

`description`

Either an item in a bullet or numbered list, or the definition of a *`term`*.

Example:

```
public class MyClass
{
    /// <summary>Here is an example of a bulleted list:
    /// <list type="bullet">
    /// <item>
    /// <description>Item 1.</description>
    /// </item>
    /// <item>
    /// <description>Item 2.</description>
    /// </item>
    /// </list>
    /// </summary>
    public static void Main () {
        // ...
    }
}
```


www.it-ebooks.info

A.2.7 <para>

This tag is for use inside other tags, such as <summary> (§A.2.11) or <returns> (§A.2.12), and permits structure to be added to text.

Syntax:

```
<para>content</para>
```

where

content

The text of the paragraph.

Example:

```
/// <summary>This is the entry point of the Point class testing program.
/// <para>This program tests each method and operator, and
/// is intended to be run after any nontrivial maintenance has
/// been performed on the Point class.</para></summary>
public static void Main() {
    // ...
}
```

A.2.8 <param>

This tag is used to describe a parameter for a method, constructor, or indexer.

Syntax:

```
<param name="name">description</param>
```

where

name

The name of the parameter.

description

A description of the parameter.

Example:

```
/// <summary>This method changes the point's location to
///   the given coordinates.</summary>
/// <param name="xor">the new x-coordinate.</param>
/// <param name="yor">the new y-coordinate.</param>
public void Move(int xor, int yor) {
    X = xor;
    Y = yor;
}
```


www.it-ebooks.info

A.2.9 <paramref>

This tag is used to indicate that a word is a parameter. The documentation file can be processed to format this parameter in some distinct way.

Syntax:

```
<paramref name="name"/>
```

where

name

The name of the parameter.

Example:

```
/// <summary>This constructor initializes the new point to
///   (<paramref name="xor"/>,<paramref name="yor"/>).</summary>
/// <param name="xor">the new point's x-coordinate.</param>
/// <param name="yor">the new point's y-coordinate.</param>
public Point(int xor, int yor) {
    X = xor;
    Y = yor;
}
```

A.2.10 <permission>

This tag allows the security accessibility of a member to be documented.

Syntax:

```
<permission cref="member">description</permission>
```

where

cref="member"

The name of a member. The documentation generator checks that the given code element exists and translates *member* to the canonical element name in the documentation file.

description

A description of the access to the member.

Example:

```
/// <permission cref="System.Security.PermissionSet">Everyone can
/// access this method.</permission>
public static void Test() {
    // ...
}
```


www.it-ebooks.info

A.2.11 <remark>

This tag is used to specify extra information about a type. (Use `<summary>` (§A.2.15) to describe the type itself and the members of a type.)

Syntax:

```
<remark>description</remark>
```

where

description

The text of the remark.

Example:

```
/// <summary>Class <c>Point</c> models a point in a
/// two-dimensional plane.</summary>
/// <remark>Uses polar coordinates</remark>
public class Point
{
    // ...
}
```

A.2.12 <returns>

This tag is used to describe the return value of a method.

Syntax:

```
<returns>description</returns>
```

where

description

A description of the return value.

Example:

```
/// <summary>Report a point's location as a string.</summary>
/// <returns>A string representing a point's location, in the form (x,y),
///   without any leading, trailing, or embedded whitespace.</returns>
public override string ToString() {
    return "(" + X + "," + Y + ")";
}
```


www.it-ebooks.info

A.2.13 <see>

This tag allows a link to be specified within text. Use `<seealso>` (§A.2.14) to indicate text that is to appear in a *See Also* section.

Syntax:

```
<see cref="member"/>
```

where

`cref="member"`

The name of a member. The documentation generator checks that the given code element exists and changes *member* to the element name in the generated documentation file.

Example:

```
/// <summary>This method changes the point's location to
///   the given coordinates.</summary>
/// <see cref="Translate"/>
public void Move(int xor, int yor) {
    X = xor;
    Y = yor;
}
/// <summary>This method changes the point's location by
///   the given x- and y-offsets.
/// </summary>
/// <see cref="Move"/>
public void Translate(int xor, int yor) {
    X += xor;
    Y += yor;
}
```

A.2.14 <seealso>

This tag allows an entry to be generated for the *See Also* section. Use `<see>` (§A.2.13) to specify a link from within text.

Syntax:

```
<seealso cref="member"/>
```

where

`cref="member"`

The name of a member. The documentation generator checks that the given code element exists and changes *member* to the element name in the generated documentation file.

www.it-ebooks.info

Example:

```
/// <summary>This method determines whether two points have the same
///   location.</summary>
/// <seealso cref="operator=="/>
/// <seealso cref="operator!="/>
public override bool Equals(object o) {
    // ...
}
```

A.2.15 <summary>

This tag can be used to describe a type or a member of a type. Use `<remark>` (§A.2.11) to describe the type itself.

Syntax:

```
<summary>description</summary>
```

where

description

A summary of the type or member.

Example:

```
/// <summary>This constructor initializes the new point to (0,0).</summary>
public Point() : this(0,0) {
}
```

A.2.16 <value>

This tag allows a property to be described.

Syntax:

```
<value>property description</value>
```

where

property description

A description for the property.

Example:

```
/// <value>Property <c>X</c> represents the point's x-coordinate.</value>
public int X
{
    get { return x; }
    set { x = value; }
}
```


www.it-ebooks.info

A.2.17 <typeparam>

This tag is used to describe a generic type parameter for a class, struct, interface, delegate, or method.

Syntax:

```
<typeparam name="name">description</typeparam>
```

where

name

The name of the type parameter.

description

A description of the type parameter.

Example:

```
/// <summary>A generic list class.</summary>
/// <typeparam name="T">The type stored by the list.</typeparam>
public class MyList<T> {
    ...
}
```

A.2.18 <typeparamref>

This tag is used to indicate that a word is a type parameter. The documentation file can be processed to format this type parameter in some distinct way.

Syntax:

```
<typeparamref name="name"/>
```

where

name

The name of the type parameter.

Example:

```
/// <summary>This method fetches data and returns a
///    list of <typeparamref name="T"> "/>"> .</summary>
/// <param name="string">query to execute</param>

public List<T> FetchData<T>(string query) {
    ...
}
```


www.it-ebooks.info

A.3 Processing the Documentation File

The documentation generator generates an ID string for each element in the source code that is tagged with a documentation comment. This ID string uniquely identifies a source element. A documentation viewer can use an ID string to identify the corresponding metadata or reflection item to which the documentation applies.

The documentation file is not a hierarchical representation of the source code; rather, it is a flat list with a generated ID string for each element.

A.3.1 ID String Format

The documentation generator observes the following rules when it generates the ID strings:

- No white space is placed in the string.
- The first part of the string identifies the kind of member being documented, via a single character followed by a colon. The following table lists the kinds of members defined.

Character	Description
E	Event
F	Field
M	Method (including constructors, destructors, and operators)
N	Namespace
P	Property (including indexers)
T	Type (such as class, delegate, enum, interface, and struct)
!	Error string; the rest of the string provides information about the error. For example, the documentation generator generates error information for links that cannot be resolved.

- The second part of the string is the fully qualified name of the element, starting at the root of the namespace. The name of the element, its enclosing type(s), and namespace are separated by periods. If the name of the item itself has periods, they are replaced by # (U+0023) characters. (It is assumed that no element has this character in its name.)

www.it-ebooks.info

- For methods and properties with arguments, the argument list follows, enclosed in parentheses. For those without arguments, the parentheses are omitted. The arguments are separated by commas. The encoding of each argument is the same as a CLI signature, as follows:
  - Arguments are represented by their documentation name, which is based on their fully qualified name, modified as follows:
    - Arguments that represent generic types have an appended "`" character followed by the number of type parameters.
    - Arguments having the `out` or `ref` modifier have an `@` following their type name. Arguments passed by value or via `params` have no special notation.
    - Arguments that are arrays are represented as `[` *lowerbound* `:` *size* `,` … `,` *lowerbound* `:` *size* `]` where the number of commas is the rank less 1, and the lower bounds and size of each dimension, if known, are represented in decimal. If a lower bound or size is not specified, it is omitted. If the lower bound and size for a particular dimension are omitted, the "`:`" is omitted as well. Jagged arrays are represented by one "`[]`" per level.
    - Arguments that have pointer types other than void are represented using a `*` following the type name. A void pointer is represented using a type name of `System.Void`.
    - Arguments that refer to generic type parameters defined on types are encoded using the "`" character followed by the zero-based index of the type parameter.
    - Arguments that use generic type parameters defined in methods use a double-backtick "``" instead of the "`" used for types.
    - Arguments that refer to constructed generic types are encoded using the generic type, followed by "{", followed by a comma-separated list of type arguments, followed by "}".

A.3.2 ID String Examples

The following examples each show a fragment of C# code, along with the ID string produced from each source element capable of having a documentation comment:

- Types are represented using their fully qualified name, augmented with generic information:

```
enum Color { Red, Blue, Green }
namespace Acme
{
    interface IProcess {...}
    struct ValueType {...}
```


www.it-ebooks.info

```
    class Widget: IProcess
    {
        public class NestedClass {...}
        public interface IMenuItem {...}
        public delegate void Del(int i);
        public enum Direction { North, South, East, West }
    }
    class MyList<T>
    {
        class Helper<U,V> {...}
    }
}

"T:Color"
"T:Acme.IProcess"
"T:Acme.ValueType"
"T:Acme.Widget"
"T:Acme.Widget.NestedClass"
"T:Acme.Widget.IMenuItem"
"T:Acme.Widget.Del"
"T:Acme.Widget.Direction"
"T:Acme.MyList`1"
"T:Acme.MyList`1.Helper`2"
```

- Fields are represented by their fully qualified name:

```
namespace Acme
{
    struct ValueType
    {
        private int total;
    }
    class Widget: IProcess
    {
        public class NestedClass
        {
            private int value;
        }
        private string message;
        private static Color defaultColor;
        private const double PI = 3.14159;
        protected readonly double monthlyAverage;
        private long[] array1;
        private Widget[,] array2;
        private unsafe int *pCount;
        private unsafe float **ppValues;
    }
}
```


www.it-ebooks.info

```
"F:Acme.ValueType.total"
"F:Acme.Widget.NestedClass.value"
"F:Acme.Widget.message"
"F:Acme.Widget.defaultColor"
"F:Acme.Widget.PI"
"F:Acme.Widget.monthlyAverage"
"F:Acme.Widget.array1"
"F:Acme.Widget.array2"
"F:Acme.Widget.pCount"
"F:Acme.Widget.ppValues"
```

- Constructors.

```
namespace Acme
{
    class Widget: IProcess
    {
        static Widget() {...}
        public Widget() {...}
        public Widget(string s) {...}
    }
}
```

```
"M:Acme.Widget.#cctor"
"M:Acme.Widget.#ctor"
"M:Acme.Widget.#ctor(System.String)"
```

- Destructors.

```
namespace Acme
{
    class Widget: IProcess
    {
        ~Widget() {...}
    }
}
```

```
"M:Acme.Widget.Finalize"
```

- Methods.

```
namespace Acme
{
    struct ValueType
    {
        public void M(int i) {...}
    }
    class Widget: IProcess
    {
        public class NestedClass
        {
            public void M(int i) {...}
        }
```


www.it-ebooks.info

```
        public static void M0() {...}
        public void M1(char c, out float f, ref ValueType v) {...}
        public void M2(short[] x1, int[,] x2, long[][] x3) {...}
        public void M3(long[][] x3, Widget[][,,] x4) {...}
        public unsafe void M4(char *pc, Color **pf) {...}
        public unsafe void M5(void *pv, double *[][,] pd) {...}
        public void M6(int i, params object[] args) {...}
    }
    class MyList<T>
    {
        public void Test(T t) { }
    }
    class UseList
    {
        public void Process(MyList<int> list) { }
        public MyList<T> GetValues<T>(T inputValue) { return null; }
    }
}

"M:Acme.ValueType.M(System.Int32)"
"M:Acme.Widget.NestedClass.M(System.Int32)"
"M:Acme.Widget.M0"
"M:Acme.Widget.M1(System.Char,System.Single@,Acme.ValueType@)"
"M:Acme.Widget.M2(System.Int16[],System.Int32[0:,0:],System.Int64[][])"
"M:Acme.Widget.M3(System.Int64[][],Acme.Widget[0:,0:,0:][])"
"M:Acme.Widget.M4(System.Char*,Color**)"
"M:Acme.Widget.M5(System.Void*,System.Double*[0:,0:][])"
"M:Acme.Widget.M6(System.Int32,System.Object[])"
"M:Acme.MyList`1.Test(`0)"
"M:Acme.UseList.Process(Acme.MyList{System.Int32})"
"M:Acme.UseList.GetValues``(``0)"
```

- Properties and indexers.

```
namespace Acme
{
    class Widget: IProcess
    {
        public int Width { get {...} set {...} }
        public int this[int i] { get {...} set {...} }
        public int this[string s, int i] { get {...} set {...} }
    }
}

"P:Acme.Widget.Width"
"P:Acme.Widget.Item(System.Int32)"
"P:Acme.Widget.Item(System.String,System.Int32)"
```

- Events.

```
namespace Acme
{
    class Widget: IProcess
    {
```


www.it-ebooks.info

```
        public event Del AnEvent;
    }
}

"E:Acme.Widget.AnEvent"
```

- Unary operators.

```
namespace Acme
{
    class Widget: IProcess
    {
        public static Widget operator+(Widget x) {...}
    }
}

"M:Acme.Widget.op_UnaryPlus(Acme.Widget)"
```

The complete set of unary operator function names used is as follows: `op_UnaryPlus`, `op_UnaryNegation`, `op_LogicalNot`, `op_OnesComplement`, `op_Increment`, `op_Decrement`, `op_True`, and `op_False`.

- Binary operators.

```
namespace Acme
{
    class Widget: IProcess
    {
        public static Widget operator+(Widget x1, Widget x2) {...}
    }
}

"M:Acme.Widget.op_Addition(Acme.Widget,Acme.Widget)"
```

The complete set of binary operator function names used is as follows: `op_Addition`, `op_Subtraction`, `op_Multiply`, `op_Division`, `op_Modulus`, `op_BitwiseAnd`, `op_BitwiseOr`, `op_ExclusiveOr`, `op_LeftShift`, `op_RightShift`, `op_Equality`, `op_Inequality`, `op_LessThan`, `op_LessThanOrEqual`, `op_GreaterThan`, and `op_GreaterThanOrEqual`.

- Conversion operators have a trailing "~" followed by the return type.

```
namespace Acme
{
    class Widget: IProcess
    {
        public static explicit operator int(Widget x) {...}
        public static implicit operator long(Widget x) {...}
    }
}

"M:Acme.Widget.op_Explicit(Acme.Widget)~System.Int32"
"M:Acme.Widget.op_Implicit(Acme.Widget)~System.Int64"
```


www.it-ebooks.info

A.4 An Example

A.4.1 C# Source Code

The following example shows the source code of a `Point` class:

```
namespace Graphics
{

    /// <summary>Class <c>Point</c> models a point in a two-dimensional plane.
    /// </summary>
    public class Point
    {
        /// <summary>Instance variable <c>x</c> represents the point's
        /// x-coordinate.</summary>
        private int x;
        /// <summary>Instance variable <c>y</c> represents the point's
        /// y-coordinate.</summary>
        private int y;
        /// <value>Property <c>X</c> represents the point's x-coordinate.</value>
        public int X
        {
            get { return x; }
            set { x = value; }
        }
        /// <value>Property <c>Y</c> represents the point's y-coordinate.</value>
        public int Y
        {
            get { return y; }
            set { y = value; }
        }
        /// <summary>This constructor initializes the new point to
        /// (0,0).</summary>
        public Point() : this(0, 0) { }
        /// <summary>This constructor initializes the new point to
        /// (<paramref name="xor"/>,<paramref name="yor"/>).</summary>
        /// <param><c>xor</c> is the new point's x-coordinate.</param>
        /// <param><c>yor</c> is the new point's y-coordinate.</param>
        public Point(int xor, int yor)
        {
            X = xor;
            Y = yor;
        }
        /// <summary>This method changes the point's location to
        /// the given coordinates.</summary>
        /// <param><c>xor</c> is the new x-coordinate.</param>
        /// <param><c>yor</c> is the new y-coordinate.</param>
        /// <see cref="Translate"/>
        public void Move(int xor, int yor)
        {
            X = xor;
            Y = yor;
        }
```


www.it-ebooks.info

```
/// <summary>This method changes the point's location by
/// the given x- and y-offsets.
/// <example>For example:
/// <code>
/// Point p = new Point(3,5);
/// p.Translate(-1,3);
/// </code>
/// results in <c>p</c>'s having the value (2,8).
/// </example>
/// </summary>
/// <param><c>xor</c> is the relative x-offset.</param>
/// <param><c>yor</c> is the relative y-offset.</param>
/// <see cref="Move"/>
public void Translate(int xor, int yor)
{
    X += xor;
    Y += yor;
}
/// <summary>This method determines whether two points have the same
/// location.</summary>
/// <param><c>o</c> is the object to be compared to the current object.
/// </param>
/// <returns>True if the points have the same location and they have
/// the exact same type; otherwise, false.</returns>
/// <seealso cref="operator=="/>
/// <seealso cref="operator!="/>
public override bool Equals(object o)
{
    if (o == null)
    {
        return false;
    }
    if (this == o)
    {
        return true;
    }
    if (GetType() == o.GetType())
    {
        Point p = (Point)o;
        return (X == p.X) && (Y == p.Y);
    }
    return false;
}
/// <summary>Report a point's location as a string.</summary>
/// <returns>A string representing a point's location, in the form (x,y),
/// without any leading, trailing, or embedded whitespace.</returns>
public override string ToString()
{
    return "(" + X + "," + Y + ")";
}
/// <summary>This operator determines whether two points have the same
/// location.</summary>
/// <param><c>p1</c> is the first point to be compared.</param>
```


www.it-ebooks.info

```
        /// <param><c>p2</c> is the second point to be compared.</param>
        /// <returns>True if the points have the same location and they have
        /// the exact same type; otherwise, false.</returns>
        /// <seealso cref="Equals"/>
        /// <seealso cref="operator!="/>
        public static bool operator ==(Point p1, Point p2)
        {
            if ((object)p1 == null || (object)p2 == null)
            {
                return false;
            }

            if (p1.GetType() == p2.GetType())
            {
                return (p1.X == p2.X) && (p1.Y == p2.Y);
            }

            return false;
        }
        /// <summary>This operator determines whether two points have the same
        /// location.</summary>
        /// <param><c>p1</c> is the first point to be compared.</param>
        /// <param><c>p2</c> is the second point to be compared.</param>
        /// <returns>True if the points do not have the same location and the
        /// exact same type; otherwise, false.</returns>
        /// <seealso cref="Equals"/>
        /// <seealso cref="operator=="/>
        public static bool operator !=(Point p1, Point p2)
        {
            return !(p1 == p2);
        }
        /// <summary>This is the entry point of the Point class testing
        /// program.
        /// <para>This program tests each method and operator, and
        /// is intended to be run after any nontrivial maintenance has
        /// been performed on the Point class.</para></summary>
        public static void Main()
        {
            // Class test code goes here
        }
    }
}
```


A.4.2 Resulting XML

Here is the output produced by one documentation generator when given the source code for class `Point`, shown above:

```
<?xml version="1.0"?>
<doc>
  <assembly>
    <name>Point</name>
  </assembly>
```


www.it-ebooks.info

```
<members>
  <member name="T:Graphics.Point">
    <summary>
      Class <c>Point</c> models a point in a two-dimensional
      plane.
    </summary>
  </member>
  <member name="F:Graphics.Point.x">
    <summary>
      Instance variable <c>x</c> represents the point's
      x-coordinate.
    </summary>
  </member>
  <member name="F:Graphics.Point.y">
    <summary>
      Instance variable <c>y</c> represents the point's
      y-coordinate.
    </summary>
  </member>
  <member name="M:Graphics.Point.#ctor">
    <summary>
      This constructor initializes the new point to
      (0,0).
    </summary>
  </member>
  <member name="M:Graphics.Point.#ctor(System.Int32,System.Int32)">
    <summary>
      This constructor initializes the new point to
      (<paramref name="xor"/>,<paramref name="yor"/>).
    </summary>
    <param>
      <c>xor</c> is the new point's x-coordinate.
    </param>
    <param>
      <c>yor</c> is the new point's y-coordinate.
    </param>
  </member>
  <member name="M:Graphics.Point.Move(System.Int32,System.Int32)">
    <summary>
      This method changes the point's location to
      the given coordinates.
    </summary>
    <param>
      <c>xor</c> is the new x-coordinate.
    </param>
    <param>
      <c>yor</c> is the new y-coordinate.
    </param>
    <see cref="M:Graphics.Point.Translate(System.Int32,System.Int32)"/>
  </member>
```


www.it-ebooks.info

```
    <member
        name="M:Graphics.Point.Translate(System.Int32,System.Int32)">
      <summary>
        This method changes the point's location by
        the given x- and y-offsets.
        <example>
          For example:
          <code>
            Point p = new Point(3,5);
            p.Translate(-1,3);
          </code>
          results in <c>p</c>'s having the value (2,8).
        </example>
      </summary>
      <param>
        <c>xor</c> is the relative x-offset.
      </param>
      <param>
        <c>yor</c> is the relative y-offset.
      </param>
      <see cref="M:Graphics.Point.Move(System.Int32,System.Int32)"/>
    </member>
    <member name="M:Graphics.Point.Equals(System.Object)">
      <summary>
        This method determines whether two points have the same
        location.
      </summary>
      <param>
        <c>o</c> is the object to be compared to the current
        object.
      </param>
      <returns>
        True if the points have the same location and they have
        the exact same type; otherwise, false.
      </returns>
      <seealso
cref="M:Graphics.Point.op_Equality(Graphics.Point,Graphics.Point)"/>
      <seealso
cref="M:Graphics.Point.op_Inequality(Graphics.Point,Graphics.Point)"/>
    </member>
    <member name="M:Graphics.Point.ToString">
      <summary>Report a point's location as a string.</summary>
      <returns>
        A string representing a point's location, in the form
        (x,y),
        without any leading, trailing, or embedded whitespace.
      </returns>
    </member>
    <member
   name="M:Graphics.Point.op_Equality(Graphics.Point,Graphics.Point)">
      <summary>
        This operator determines whether two points have the
        same
        location.
```


www.it-ebooks.info

```
      </summary>
      <param>
        <c>p1</c> is the first point to be compared.
      </param>
      <param>
        <c>p2</c> is the second point to be compared.
      </param>
      <returns>
        True if the points have the same location and they have
        the exact same type; otherwise, false.
      </returns>
      <seealso cref="M:Graphics.Point.Equals(System.Object)"/>
      <seealso
     cref="M:Graphics.Point.op_Inequality(Graphics.Point,Graphics.Point)"/>
    </member>
    <member
  name="M:Graphics.Point.op_Inequality(Graphics.Point,Graphics.Point)">
      <summary>
        This operator determines whether two points have the
        same
        location.
      </summary>
      <param>
        <c>p1</c> is the first point to be compared.
      </param>
      <param>
        <c>p2</c> is the second point to be compared.
      </param>
      <returns>
        True if the points do not have the same location and
        the
        exact same type; otherwise, false.
      </returns>
      <seealso cref="M:Graphics.Point.Equals(System.Object)"/>
      <seealso
cref="M:Graphics.Point.op_Equality(Graphics.Point,Graphics.Point)"/>
    </member>
    <member name="M:Graphics.Point.Main">
      <summary>
        This is the entry point of the Point class testing
        program.
        <para>
          This program tests each method and operator, and
          is intended to be run after any nontrivial maintenance has
          been performed on the Point class.
        </para>
      </summary>
    </member>
    <member name="P:Graphics.Point.X">
      <value>
        Property <c>X</c> represents the point's
        x-coordinate.
      </value>
    </member>
```


www.it-ebooks.info

```
    <member name="P:Graphics.Point.Y">
      <value>
        Property <c>Y</c> represents the point's
        y-coordinate.
      </value>
    </member>
  </members>
</doc>
```


www.it-ebooks.info

B. Grammar

This appendix contains summaries of the lexical and syntactic grammars found in the main document, and of the grammar extensions for unsafe code. Grammar productions appear here in the same order that they appear in the main document.

B.1 Lexical Grammar

input:
    input-section$_{opt}$

input-section:
    input-section-part
    input-section *input-section-part*

input-section-part:
    input-elements$_{opt}$ *new-line*
    pp-directive

input-elements:
    input-element
    input-elements *input-element*

input-element:
    whitespace
    comment
    token

B.1.1 Line Terminators

new-line:
    Carriage return character (`U+000D`)
    Line feed character (`U+000A`)
    Carriage return character (`U+000D`) followed by line feed character (`U+000A`)
    Next line character (`U+0085`)
    Line separator character (`U+2028`)
    Paragraph separator character (`U+2029`)

www.it-ebooks.info

B.1.2 Comments

comment:
    single-line-comment
    delimited-comment

single-line-comment:
    `//` *input-characters*$_{opt}$

input-characters:
    input-character
    input-characters *input-character*

input-character:
    Any Unicode character except a *new-line-character*

new-line-character:
    Carriage return character (`U+000D`)
    Line feed character (`U+000A`)
    Next line character (`U+0085`)
    Line separator character (`U+2028`)
    Paragraph separator character (`U+2029`)

delimited-comment:
    `/*` *delimited-comment-text*$_{opt}$ *asterisks* `/`

delimited-comment-text:
    delimited-comment-section
    delimited-comment-text *delimited-comment-section*

delimited-comment-section:
    `/`
    asterisks$_{opt}$ *not-slash-or-asterisk*

asterisks:
    `*`
    asterisks `*`

not-slash-or-asterisk:
    Any Unicode character except `/` or `*`

www.it-ebooks.info

B.1.3 White Space

whitespace:
Any character with Unicode class Zs
Horizontal tab character (`U+0009`)
Vertical tab character (`U+000B`)
Form feed character (`U+000C`)

B.1.4 Tokens

token:
identifier
keyword
integer-literal
real-literal
character-literal
string-literal
operator-or-punctuator

B.1.5 Unicode Character Escape Sequences

unicode-escape-sequence:
`\u` *hex-digit hex-digit hex-digit hex-digit*
`\U` *hex-digit hex-digit hex-digit hex-digit hex-digit hex-digit hex-digit hex-digit*

B.1.6 Identifiers

identifier:
available-identifier
`@` *identifier-or-keyword*

available-identifier:
An *identifier-or-keyword* that is not a *keyword*

identifier-or-keyword:
identifier-start-character identifier-part-characters$_{opt}$

identifier-start-character:
letter-character
_ (the underscore character `U+005F`)

identifier-part-characters:
identifier-part-character
identifier-part-characters identifier-part-character

www.it-ebooks.info

identifier-part-character:
letter-character
decimal-digit-character
connecting-character
combining-character
formatting-character

letter-character:
A Unicode character of classes Lu, Ll, Lt, Lm, Lo, or Nl
A *unicode-escape-sequence* representing a character of classes Lu, Ll, Lt, Lm, Lo, or Nl

combining-character:
A Unicode character of classes Mn or Mc
A *unicode-escape-sequence* representing a character of class Mn or Mc

decimal-digit-character:
A Unicode character of the class Nd
A *unicode-escape-sequence* representing a character of the class Nd

connecting-character:
A Unicode character of the class Pc
A *unicode-escape-sequence* representing a character of the class Pc

formatting-character:
A Unicode character of the class Cf
A *unicode-escape-sequence* representing a character of the class Cf

B.1.7 Keywords

keyword: one of

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


www.it-ebooks.info

B.1.8 Literals

literal:
    boolean-literal
    integer-literal
    real-literal
    character-literal
    string-literal
    null-literal

boolean-literal:
    `true`
    `false`

integer-literal:
    decimal-integer-literal
    hexadecimal-integer-literal

decimal-integer-literal:
    decimal-digits *integer-type-suffix*$_{opt}$

decimal-digits:
    decimal-digit
    decimal-digits *decimal-digit*

decimal-digit: one of
    `0` `1` `2` `3` `4` `5` `6` `7` `8` `9`

integer-type-suffix: one of
    `U` `u` `L` `l` `UL` `Ul` `uL` `ul` `LU` `Lu` `lU` `lu`

hexadecimal-integer-literal:
    `0x` *hex-digits* *integer-type-suffix*$_{opt}$
    `0X` *hex-digits* *integer-type-suffix*$_{opt}$

hex-digits:
    hex-digit
    hex-digits *hex-digit*

hex-digit: one of
    `0` `1` `2` `3` `4` `5` `6` `7` `8` `9` `A` `B` `C` `D` `E` `F` `a` `b` `c` `d` `e` `f`

www.it-ebooks.info

real-literal:
    decimal-digits `.` *decimal-digits* *exponent-part*$_{opt}$ *real-type-suffix*$_{opt}$
    `.` *decimal-digits* *exponent-part*$_{opt}$ *real-type-suffix*$_{opt}$
    decimal-digits *exponent-part* *real-type-suffix*$_{opt}$
    decimal-digits *real-type-suffix*

exponent-part:
    `e` *sign*$_{opt}$ *decimal-digits*
    `E` *sign*$_{opt}$ *decimal-digits*

sign: one of
    `+` `-`

real-type-suffix: one of
    `F` `f` `D` `d` `M` `m`

character-literal:
    `'` *character* `'`

character:
    single-character
    simple-escape-sequence
    hexadecimal-escape-sequence
    unicode-escape-sequence

single-character:
    Any character except `'` (U+0027), `\` (U+005C), and *new-line-character*

simple-escape-sequence: one of
    `\'` `\"` `\\` `\0` `\a` `\b` `\f` `\n` `\r` `\t` `\v`

hexadecimal-escape-sequence:
    `\x` *hex-digit* *hex-digit*$_{opt}$ *hex-digit*$_{opt}$ *hex-digit*$_{opt}$

string-literal:
    regular-string-literal
    verbatim-string-literal

regular-string-literal:
    `"` *regular-string-literal-characters*$_{opt}$ `"`

regular-string-literal-characters:
    regular-string-literal-character
    regular-string-literal-characters *regular-string-literal-character*

www.it-ebooks.info

regular-string-literal-character:
    single-regular-string-literal-character
    simple-escape-sequence
    hexadecimal-escape-sequence
    unicode-escape-sequence

single-regular-string-literal-character:
    Any character except " (U+0022), \ (U+005C), and *new-line-character*

verbatim-string-literal:
    @" *verbatim-string-literal-characters*$_{opt}$ "

verbatim-string-literal-characters:
    verbatim-string-literal-character
    verbatim-string-literal-characters verbatim-string-literal-character

verbatim-string-literal-character:
    single-verbatim-string-literal-character
    quote-escape-sequence

single-verbatim-string-literal-character:
    Any character except "

quote-escape-sequence:
    ""

null-literal:
    null

B.1.9 Operators and Punctuators

operator-or-punctuator: one of

```
{     }     [     ]     (     )     .     ,     :     ;
+     -     *     /     %     &     |     ^     !     ~
=     <     >     ?     ??    ::    ++    --    &&    ||
->    ==    !=    <=    >=    +=    -=    *=    /=    %=
&=    |=    ^=    <<    <<=   =>
```

right-shift:
    >|>

right-shift-assignment:
    >|>=

www.it-ebooks.info

B.1.10 Preprocessing Directives

pp-directive:
pp-declaration
pp-conditional
pp-line
pp-diagnostic
pp-region
pp-pragma

conditional-symbol:
Any *identifier-or-keyword* except `true` or `false`

pp-expression:
whitespace$_{opt}$ *pp-or-expression* *whitespace*$_{opt}$

pp-or-expression:
pp-and-expression
pp-or-expression *whitespace*$_{opt}$ `||` *whitespace*$_{opt}$ *pp-and-expression*

pp-and-expression:
pp-equality-expression
pp-and-expression *whitespace*$_{opt}$ `&&` *whitespace*$_{opt}$ *pp-equality-expression*

pp-equality-expression:
pp-unary-expression
pp-equality-expression *whitespace*$_{opt}$ `==` *whitespace*$_{opt}$ *pp-unary-expression*
pp-equality-expression *whitespace*$_{opt}$ `!=` *whitespace*$_{opt}$ *pp-unary-expression*

pp-unary-expression:
pp-primary-expression
`!` *whitespace*$_{opt}$ *pp-unary-expression*

pp-primary-expression:
`true`
`false`
conditional-symbol
`(` *whitespace*$_{opt}$ *pp-expression* *whitespace*$_{opt}$ `)`

pp-declaration:
whitespace$_{opt}$ `#` *whitespace*$_{opt}$ `define` *whitespace* *conditional-symbol* *pp-new-line*
whitespace$_{opt}$ `#` *whitespace*$_{opt}$ `undef` *whitespace* *conditional-symbol* *pp-new-line*

www.it-ebooks.info

pp-new-line:
whitespace$_{opt}$ *single-line-comment*$_{opt}$ *new-line*

pp-conditional:
pp-if-section *pp-elif-sections*$_{opt}$ *pp-else-section*$_{opt}$ *pp-endif*

pp-if-section:
whitespace$_{opt}$ `#` *whitespace*$_{opt}$ `if` *whitespace* *pp-expression* *pp-new-line*
conditional-section$_{opt}$

pp-elif-sections:
pp-elif-section
pp-elif-sections *pp-elif-section*

pp-elif-section:
whitespace$_{opt}$ `#` *whitespace*$_{opt}$ `elif` *whitespace* *pp-expression* *pp-new-line*
conditional-section$_{opt}$

pp-else-section:
whitespace$_{opt}$ `#` *whitespace*$_{opt}$ `else` *pp-new-line* *conditional-section*$_{opt}$

pp-endif:
whitespace$_{opt}$ `#` *whitespace*$_{opt}$ `endif` *pp-new-line*

conditional-section:
input-section
skipped-section

skipped-section:
skipped-section-part
skipped-section *skipped-section-part*

skipped-section-part:
skipped-characters$_{opt}$ *new-line*
pp-directive

skipped-characters:
whitespace$_{opt}$ *not-number-sign* *input-characters*$_{opt}$

not-number-sign:
Any input-character except `#`

www.it-ebooks.info

pp-diagnostic:
    whitespace$_{opt}$ `#` *whitespace*$_{opt}$ `error` *pp-message*
    whitespace$_{opt}$ `#` *whitespace*$_{opt}$ `warning` *pp-message*

pp-message:
    new-line
    whitespace *input-characters*$_{opt}$ *new-line*

pp-region:
    pp-start-region *conditional-section*$_{opt}$ *pp-end-region*

pp-start-region:
    whitespace$_{opt}$ `#` *whitespace*$_{opt}$ `region` *pp-message*

pp-end-region:
    whitespace$_{opt}$ `#` *whitespace*$_{opt}$ `endregion` *pp-message*

pp-line:
    whitespace$_{opt}$ `#` *whitespace*$_{opt}$ `line` *whitespace* *line-indicator* *pp-new-line*

line-indicator:
    decimal-digits *whitespace* *file-name*
    decimal-digits
    `default`
    `hidden`

file-name:
    `"` *file-name-characters* `"`

file-name-characters:
    file-name-character
    file-name-characters *file-name-character*

file-name-character:
    Any *input-character* except "

pp-pragma:
    whitespace$_{opt}$ `#` *whitespace*$_{opt}$ `pragma` *whitespace* *pragma-body* *pp-new-line*

pragma-body:
    pragma-warning-body

pragma-warning-body:
    `warning` *whitespace* *warning-action*
    `warning` *whitespace* *warning-action* *whitespace* *warning-list*

www.it-ebooks.info

warning-action:
    `disable`
    `restore`

warning-list:
    decimal-digits
    warning-list *whitespace*$_{opt}$ `,` *whitespace*$_{opt}$ *decimal-digits*

B.2 Syntactic Grammar

B.2.1 Basic Concepts

namespace-name:
    namespace-or-type-name

type-name:
    namespace-or-type-name

namespace-or-type-name:
    identifier *type-argument-list*$_{opt}$
    namespace-or-type-name `.` *identifier* *type-argument-list*$_{opt}$
    qualified-alias-member

B.2.2 Types

type:
    value-type
    reference-type
    type-parameter

value-type:
    struct-type
    enum-type

struct-type:
    type-name
    simple-type
    nullable-type

simple-type:
    numeric-type
    `bool`

www.it-ebooks.info

numeric-type:
    integral-type
    floating-point-type
    `decimal`

integral-type:
    `sbyte`
    `byte`
    `short`
    `ushort`
    `int`
    `uint`
    `long`
    `ulong`
    `char`

floating-point-type:
    `float`
    `double`

nullable-type:
    non-nullable-value-type `?`

non-nullable-value-type:
    type

enum-type:
    type-name

reference-type:
    class-type
    interface-type
    array-type
    delegate-type

class-type:
    type-name
    `object`
    `dynamic`
    `string`

interface-type:
    type-name

www.it-ebooks.info

array-type:
    non-array-type *rank-specifiers*

non-array-type:
    type

rank-specifiers:
    rank-specifier
    rank-specifiers *rank-specifier*

rank-specifier:
    `[` *dim-separators*$_{opt}$ `]`

dim-separators:
    `,`
    dim-separators `,`

delegate-type:
    type-name

type-argument-list:
    `<` *type-arguments* `>`

type-arguments:
    type-argument
    type-arguments `,` *type-argument*

type-argument:
    type

type-parameter:
    identifier

B.2.3 Variables

variable-reference:
    expression

B.2.4 Expressions

argument-list:
    argument
    argument-list `,` *argument*

www.it-ebooks.info

argument:
    argument-name$_{opt}$ *argument-value*

argument-name:
    identifier `:`

argument-value:
    expression
    `ref` *variable-reference*
    `out` *variable-reference*

primary-expression:
    primary-no-array-creation-expression
    array-creation-expression

primary-no-array-creation-expression:
    literal
    simple-name
    parenthesized-expression
    member-access
    invocation-expression
    element-access
    this-access
    base-access
    post-increment-expression
    post-decrement-expression
    object-creation-expression
    delegate-creation-expression
    anonymous-object-creation-expression
    typeof-expression
    checked-expression
    unchecked-expression
    default-value-expression
    anonymous-method-expression

simple-name:
    identifier *type-argument-list*$_{opt}$

parenthesized-expression:
    `(` *expression* `)`

www.it-ebooks.info

member-access:
    primary-expression `.` *identifier type-argument-list*$_{opt}$
    predefined-type `.` *identifier type-argument-list*$_{opt}$
    qualified-alias-member `.` *identifier type-argument-list*$_{opt}$

predefined-type: one of
    `bool` `byte` `char` `decimal` `double` `float` `int` `long`
    `object` `sbyte` `short` `string` `uint` `ulong` `ushort`

invocation-expression:
    primary-expression `(` *argument-list*$_{opt}$ `)`

element-access:
    primary-no-array-creation-expression `[` *argument-list* `]`

this-access:
    `this`

base-access:
    `base` `.` *identifier*
    `base` `[` *argument-list* `]`

post-increment-expression:
    primary-expression `++`

post-decrement-expression:
    primary-expression `--`

object-creation-expression:
    `new` *type* `(` *argument-list*$_{opt}$ `)` *object-or-collection-initializer*$_{opt}$
    `new` *type object-or-collection-initializer*

object-or-collection-initializer:
    object-initializer
    collection-initializer

object-initializer:
    `{` *member-initializer-list*$_{opt}$ `}`
    `{` *member-initializer-list* `,` `}`

member-initializer-list:
    member-initializer
    member-initializer-list `,` *member-initializer*

www.it-ebooks.info

member-initializer:
    identifier `=` *initializer-value*

initializer-value:
    expression
    object-or-collection-initializer

collection-initializer:
    `{` *element-initializer-list* `}`
    `{` *element-initializer-list* `,` `}`

element-initializer-list:
    element-initializer
    element-initializer-list `,` *element-initializer*

element-initializer:
    non-assignment-expression
    `{` *expression-list* `}`

expression-list:
    expression
    expression-list `,` *expression*

array-creation-expression:
    `new` *non-array-type* `[` *expression-list* `]` *rank-specifiers*$_{opt}$ *array-initializer*$_{opt}$
    `new` *array-type* *array-initializer*
    `new` *rank-specifier* *array-initializer*

delegate-creation-expression:
    `new` *delegate-type* `(` *expression* `)`

anonymous-object-creation-expression:
    `new` *anonymous-object-initializer*

anonymous-object-initializer:
    `{` *member-declarator-list*$_{opt}$ `}`
    `{` *member-declarator-list* `,` `}`

member-declarator-list:
    member-declarator
    member-declarator-list `,` *member-declarator*

www.it-ebooks.info

member-declarator:
    simple-name
    member-access
    base-access
    identifier `=` *expression*

typeof-expression:
    `typeof` `(` *type* `)`
    `typeof` `(` *unbound-type-name* `)`
    `typeof` `(` `void` `)`

unbound-type-name:
    identifier *generic-dimension-specifier*$_{opt}$
    identifier `::` *identifier* *generic-dimension-specifier*$_{opt}$
    unbound-type-name `.` *identifier* *generic-dimension-specifier*$_{opt}$

generic-dimension-specifier:
    `<` *commas*$_{opt}$ `>`

commas:
    `,`
    commas `,`

checked-expression:
    `checked` `(` *expression* `)`

unchecked-expression:
    `unchecked` `(` *expression* `)`

default-value-expression:
    `default` `(` *type* `)`

unary-expression:
    primary-expression
    `+` *unary-expression*
    `-` *unary-expression*
    `!` *unary-expression*
    `~` *unary-expression*
    pre-increment-expression
    pre-decrement-expression
    cast-expression

www.it-ebooks.info

pre-increment-expression:
    `++` *unary-expression*

pre-decrement-expression:
    `--` *unary-expression*

cast-expression:
    `(` *type* `)` *unary-expression*

multiplicative-expression:
    unary-expression
    multiplicative-expression `*` *unary-expression*
    multiplicative-expression `/` *unary-expression*
    multiplicative-expression `%` *unary-expression*

additive-expression:
    multiplicative-expression
    additive-expression `+` *multiplicative-expression*
    additive-expression `–` *multiplicative-expression*

shift-expression:
    additive-expression
    shift-expression `<<` *additive-expression*
    shift-expression *right-shift* *additive-expression*

relational-expression:
    shift-expression
    relational-expression `<` *shift-expression*
    relational-expression `>` *shift-expression*
    relational-expression `<=` *shift-expression*
    relational-expression `>=` *shift-expression*
    relational-expression *is* *type*
    relational-expression *as* *type*

equality-expression:
    relational-expression
    equality-expression `==` *relational-expression*
    equality-expression `!=` *relational-expression*

and-expression:
    equality-expression
    and-expression `&` *equality-expression*

www.it-ebooks.info

exclusive-or-expression:
    and-expression
    exclusive-or-expression `^` *and-expression*

inclusive-or-expression:
    exclusive-or-expression
    inclusive-or-expression `|` *exclusive-or-expression*

conditional-and-expression:
    inclusive-or-expression
    conditional-and-expression `&&` *inclusive-or-expression*

conditional-or-expression:
    conditional-and-expression
    conditional-or-expression `||` *conditional-and-expression*

null-coalescing-expression:
    conditional-or-expression
    conditional-or-expression `??` *null-coalescing-expression*

conditional-expression:
    null-coalescing-expression
    null-coalescing-expression `?` *expression* `:` *expression*

lambda-expression:
    anonymous-function-signature `=>` *anonymous-function-body*

anonymous-method-expression:
    `delegate` *explicit-anonymous-function-signature*$_{opt}$ *block*

anonymous-function-signature:
    explicit-anonymous-function-signature
    implicit-anonymous-function-signature

explicit-anonymous-function-signature:
    `(` *explicit-anonymous-function-parameter-list*$_{opt}$ `)`

explicit-anonymous-function-parameter-list:
    explicit-anonymous-function-parameter
    explicit-anonymous-function-parameter-list `,` *explicit-anonymous-function-parameter*

explicit-anonymous-function-parameter:
    anonymous-function-parameter-modifier$_{opt}$ *type* *identifier*

www.it-ebooks.info

anonymous-function-parameter-modifier:
    `ref`
    `out`

implicit-anonymous-function-signature:
    `(` *implicit-anonymous-function-parameter-list*$_{opt}$ `)`
    implicit-anonymous-function-parameter

implicit-anonymous-function-parameter-list:
    implicit-anonymous-function-parameter
    implicit-anonymous-function-parameter-list `,`
        implicit-anonymous-function-parameter

implicit-anonymous-function-parameter:
    identifier

anonymous-function-body:
    expression
    block

query-expression:
    from-clause *query-body*

from-clause:
    `from` *type*$_{opt}$ *identifier* `in` *expression*

query-body:
    query-body-clauses$_{opt}$ *select-or-group-clause* *query-continuation*$_{opt}$

query-body-clauses:
    query-body-clause
    query-body-clauses *query-body-clause*

query-body-clause:
    from-clause
    let-clause
    where-clause
    join-clause
    join-into-clause
    orderby-clause

let-clause:
    `let` *identifier* `=` *expression*

where-clause:
    `where` *boolean-expression*

join-clause:
    `join` *type*$_{opt}$ *identifier* `in` *expression* `on` *expression* `equals` *expression*

www.it-ebooks.info

join-into-clause:
    `join` *type*$_{opt}$ *identifier* `in` *expression* `on` *expression* `equals` *expression*
        `into` *identifier*

orderby-clause:
    `orderby` *orderings*

orderings:
    ordering
    orderings `,` *ordering*

ordering:
    expression *ordering-direction*$_{opt}$

ordering-direction:
    `ascending`
    `descending`

select-or-group-clause:
    select-clause
    group-clause

select-clause:
    `select` *expression*

group-clause:
    `group` *expression* `by` *expression*

query-continuation:
    `into` *identifier* *query-body*

assignment:
    unary-expression *assignment-operator* *expression*

assignment-operator:
    `=`
    `+=`
    `-=`
    `*=`
    `/=`
    `%=`
    `&=`
    `|=`
    `^=`
    `<<=`
    right-shift-assignment

www.it-ebooks.info

expression:
    non-assignment-expression
    assignment

non-assignment-expression:
    conditional-expression
    lambda-expression
    query-expression

constant-expression:
    expression

boolean-expression:
    expression

B.2.5 Statements

statement:
    labeled-statement
    declaration-statement
    embedded-statement

embedded-statement:
    block
    empty-statement
    expression-statement
    selection-statement
    iteration-statement
    jump-statement
    try-statement
    checked-statement
    unchecked-statement
    lock-statement
    using-statement
    yield-statement

block:
    { *statement-list*$_{opt}$ }

statement-list:
    statement
    statement-list *statement*

www.it-ebooks.info

empty-statement:
    ;

labeled-statement:
    identifier : *statement*

declaration-statement:
    local-variable-declaration ;
    local-constant-declaration ;

local-variable-declaration:
    local-variable-type *local-variable-declarators*

local-variable-type:
    type
    var

local-variable-declarators:
    local-variable-declarator
    local-variable-declarators , *local-variable-declarator*

local-variable-declarator:
    identifier
    identifier = *local-variable-initializer*

local-variable-initializer:
    expression
    array-initializer

local-constant-declaration:
    const *type* *constant-declarators*

constant-declarators:
    constant-declarator
    constant-declarators , *constant-declarator*

constant-declarator:
    identifier = *constant-expression*

expression-statement:
    statement-expression ;

www.it-ebooks.info

statement-expression:
    invocation-expression
    object-creation-expression
    assignment
    post-increment-expression
    post-decrement-expression
    pre-increment-expression
    pre-decrement-expression

selection-statement:
    if-statement
    switch-statement

if-statement:
    `if` `(` *boolean-expression* `)` *embedded-statement*
    `if` `(` *boolean-expression* `)` *embedded-statement* `else` *embedded-statement*

switch-statement:
    `switch` `(` *expression* `)` *switch-block*

switch-block:
    `{` *switch-sections*$_{opt}$ `}`

switch-sections:
    switch-section
    switch-sections *switch-section*

switch-section:
    switch-labels *statement-list*

switch-labels:
    switch-label
    switch-labels *switch-label*

switch-label:
    `case` *constant-expression* `:`
    `default` `:`

iteration-statement:
    while-statement
    do-statement
    for-statement
    foreach-statement

www.it-ebooks.info

while-statement:
    `while` `(` *boolean-expression* `)` *embedded-statement*

do-statement:
    `do` *embedded-statement* `while` `(` *boolean-expression* `)` `;`

for-statement:
    `for` `(` *for-initializer*$_{opt}$ `;` *for-condition*$_{opt}$ `;` *for-iterator*$_{opt}$ `)` *embedded-statement*

for-initializer:
    local-variable-declaration
    statement-expression-list

for-condition:
    boolean-expression

for-iterator:
    statement-expression-list

statement-expression-list:
    statement-expression
    statement-expression-list `,` *statement-expression*

foreach-statement:
    `foreach` `(` *local-variable-type* *identifier* `in` *expression* `)` *embedded-statement*

jump-statement:
    break-statement
    continue-statement
    goto-statement
    return-statement
    throw-statement

break-statement:
    `break` `;`

continue-statement:
    `continue` `;`

goto-statement:
    `goto` *identifier* `;`
    `goto` `case` *constant-expression* `;`
    `goto` `default` `;`

www.it-ebooks.info

return-statement:
    `return` *expression*$_{opt}$ `;`

throw-statement:
    `throw` *expression*$_{opt}$ `;`

try-statement:
    `try` *block* *catch-clauses*
    `try` *block* *finally-clause*
    `try` *block* *catch-clauses* *finally-clause*

catch-clauses:
    specific-catch-clauses *general-catch-clause*$_{opt}$
    specific-catch-clauses$_{opt}$ *general-catch-clause*

specific-catch-clauses:
    specific-catch-clause
    specific-catch-clauses *specific-catch-clause*

specific-catch-clause:
    `catch` `(` *class-type* *identifier*$_{opt}$ `)` *block*

general-catch-clause:
    `catch` *block*

finally-clause:
    `finally` *block*

checked-statement:
    `checked` *block*

unchecked-statement:
    `unchecked` *block*

lock-statement:
    `lock` `(` *expression* `)` *embedded-statement*

using-statement:
    `using` `(` *resource-acquisition* `)` *embedded-statement*

resource-acquisition:
    local-variable-declaration
    expression

www.it-ebooks.info

yield-statement:
    `yield return` *expression* `;`
    `yield break ;`

B.2.6 Namespaces

compilation-unit:
    extern-alias-directives$_{opt}$ *using-directives*$_{opt}$ *global-attributes*$_{opt}$
        namespace-member-declarations$_{opt}$

namespace-declaration:
    `namespace` *qualified-identifier namespace-body* `;`$_{opt}$

qualified-identifier:
    identifier
    qualified-identifier `.` *identifier*

namespace-body:
    `{` *extern-alias-directives*$_{opt}$ *using-directives*$_{opt}$ *namespace-member-declarations*$_{opt}$ `}`

extern-alias-directives:
    extern-alias-directive
    extern-alias-directives extern-alias-directive

extern-alias-directive:
    `extern alias` *identifier* `;`

using-directives:
    using-directive
    using-directives using-directive

using-directive:
    using-alias-directive
    using-namespace-directive

using-alias-directive:
    `using` *identifier* `=` *namespace-or-type-name* `;`

using-namespace-directive:
    `using` *namespace-name* `;`

namespace-member-declarations:
    namespace-member-declaration
    namespace-member-declarations namespace-member-declaration

www.it-ebooks.info

namespace-member-declaration:
namespace-declaration
type-declaration

type-declaration:
class-declaration
struct-declaration
interface-declaration
enum-declaration
delegate-declaration

qualified-alias-member:
identifier `::` *identifier* *type-argument-list*$_{opt}$

B.2.7 Classes

class-declaration:
attributes$_{opt}$ *class-modifiers*$_{opt}$ `partial`$_{opt}$ `class` *identifier* *type-parameter-list*$_{opt}$
class-base$_{opt}$ *type-parameter-constraints-clauses*$_{opt}$ *class-body* `;`$_{opt}$

class-modifiers:
class-modifier
class-modifiers *class-modifier*

class-modifier:
`new`
`public`
`protected`
`internal`
`private`
`abstract`
`sealed`
`static`

type-parameter-list:
`<` *type-parameters* `>`

type-parameters:
attributes$_{opt}$ *type-parameter*
type-parameters `,` *attributes*$_{opt}$ *type-parameter*

www.it-ebooks.info

type-parameter:
    identifier

class-base:
    `:` *class-type*
    `:` *interface-type-list*
    `:` *class-type* `,` *interface-type-list*

interface-type-list:
    interface-type
    interface-type-list `,` *interface-type*

type-parameter-constraints-clauses:
    type-parameter-constraints-clause
    type-parameter-constraints-clauses *type-parameter-constraints-clause*

type-parameter-constraints-clause:
    `where` *type-parameter* `:` *type-parameter-constraints*

type-parameter-constraints:
    primary-constraint
    secondary-constraints
    constructor-constraint
    primary-constraint `,` *secondary-constraints*
    primary-constraint `,` *constructor-constraint*
    secondary-constraints `,` *constructor-constraint*
    primary-constraint `,` *secondary-constraints* `,` *constructor-constraint*

primary-constraint:
    class-type
    `class`
    `struct`

secondary-constraints:
    interface-type
    type-parameter
    secondary-constraints `,` *interface-type*
    secondary-constraints `,` *type-parameter*

constructor-constraint:
    `new ( )`

www.it-ebooks.info

class-body:
    `{` *class-member-declarations*$_{opt}$ `}`

class-member-declarations:
    class-member-declaration
    class-member-declarations *class-member-declaration*

class-member-declaration:
    constant-declaration
    field-declaration
    method-declaration
    property-declaration
    event-declaration
    indexer-declaration
    operator-declaration
    constructor-declaration
    destructor-declaration
    static-constructor-declaration
    type-declaration

constant-declaration:
    attributes$_{opt}$ *constant-modifiers*$_{opt}$ `const` *type* *constant-declarators* `;`

constant-modifiers:
    constant-modifier
    constant-modifiers *constant-modifier*

constant-modifier:
    `new`
    `public`
    `protected`
    `internal`
    `private`

constant-declarators:
    constant-declarator
    constant-declarators `,` *constant-declarator*

constant-declarator:
    identifier `=` *constant-expression*

www.it-ebooks.info

field-declaration:
    attributes$_{opt}$ *field-modifiers*$_{opt}$ *type* *variable-declarators* `;`

field-modifiers:
    field-modifier
    field-modifiers *field-modifier*

field-modifier:
    `new`
    `public`
    `protected`
    `internal`
    `private`
    `static`
    `readonly`
    `volatile`

variable-declarators:
    variable-declarator
    variable-declarators `,` *variable-declarator*

variable-declarator:
    identifier
    identifier `=` *variable-initializer*

variable-initializer:
    expression
    array-initializer

method-declaration:
    method-header *method-body*

method-header:
    attributes$_{opt}$ *method-modifiers*$_{opt}$ `partial`$_{opt}$ *return-type* *member-name*
        type-parameter-list$_{opt}$
            `(` *formal-parameter-list*$_{opt}$ `)` *type-parameter-constraints-clauses*$_{opt}$

method-modifiers:
    method-modifier
    method-modifiers *method-modifier*

www.it-ebooks.info

method-modifier:
    `new`
    `public`
    `protected`
    `internal`
    `private`
    `static`
    `virtual`
    `sealed`
    `override`
    `abstract`
    `extern`

return-type:
    type
    `void`

member-name:
    identifier
    interface-type `.` *identifier*

method-body:
    block
    `;`

formal-parameter-list:
    fixed-parameters
    fixed-parameters `,` *parameter-array*
    parameter-array

fixed-parameters:
    fixed-parameter
    fixed-parameters `,` *fixed-parameter*

fixed-parameter:
    $attributes_{opt}$ $parameter\text{-}modifier_{opt}$ *type* *identifier* $default\text{-}argument_{opt}$

default-argument:
    `=` *expression*

www.it-ebooks.info

parameter-modifier:
    `ref`
    `out`
    `this`

parameter-array:
    attributes$_{opt}$ `params` *array-type* *identifier*

property-declaration:
    attributes$_{opt}$ *property-modifiers*$_{opt}$ *type* *member-name* `{` *accessor-declarations* `}`

property-modifiers:
    property-modifier
    property-modifiers *property-modifier*

property-modifier:
    `new`
    `public`
    `protected`
    `internal`
    `private`
    `static`
    `virtual`
    `sealed`
    `override`
    `abstract`
    `extern`

member-name:
    identifier
    interface-type `.` *identifier*

accessor-declarations:
    get-accessor-declaration *set-accessor-declaration*$_{opt}$
    set-accessor-declaration *get-accessor-declaration*$_{opt}$

get-accessor-declaration:
    attributes$_{opt}$ *accessor-modifier*$_{opt}$ `get` *accessor-body*

set-accessor-declaration:
    attributes$_{opt}$ *accessor-modifier*$_{opt}$ `set` *accessor-body*

www.it-ebooks.info

accessor-modifier:
    `protected`
    `internal`
    `private`
    `protected` `internal`
    `internal` `protected`

accessor-body:
    block
    `;`

event-declaration:
    attributes$_{opt}$ *event-modifiers*$_{opt}$ `event` *type* *variable-declarators* `;`
    attributes$_{opt}$ *event-modifiers*$_{opt}$ `event` *type* *member-name*
        `{` *event-accessor-declarations* `}`

event-modifiers:
    event-modifier
    event-modifiers *event-modifier*

event-modifier:
    `new`
    `public`
    `protected`
    `internal`
    `private`
    `static`
    `virtual`
    `sealed`
    `override`
    `abstract`
    `extern`

event-accessor-declarations:
    add-accessor-declaration *remove-accessor-declaration*
    remove-accessor-declaration *add-accessor-declaration*

add-accessor-declaration:
    attributes$_{opt}$ `add` *block*

remove-accessor-declaration:
    attributes$_{opt}$ `remove` *block*

www.it-ebooks.info

indexer-declaration:
    attributes$_{opt}$ *indexer-modifiers*$_{opt}$ *indexer-declarator* `{` *accessor-declarations* `}`

indexer-modifiers:
    indexer-modifier
    indexer-modifiers *indexer-modifier*

indexer-modifier:
    `new`
    `public`
    `protected`
    `internal`
    `private`
    `virtual`
    `sealed`
    `override`
    `abstract`
    `extern`

indexer-declarator:
    type `this` `[` *formal-parameter-list* `]`
    type *interface-type* `.` `this` `[` *formal-parameter-list* `]`

operator-declaration:
    attributes$_{opt}$ *operator-modifiers* *operator-declarator* *operator-body*

operator-modifiers:
    operator-modifier
    operator-modifiers *operator-modifier*

operator-modifier:
    `public`
    `static`
    `extern`

operator-declarator:
    unary-operator-declarator
    binary-operator-declarator
    conversion-operator-declarator

unary-operator-declarator:
    type `operator` *overloadable-unary-operator* `(` *type* *identifier* `)`

overloadable-unary-operator: one of
    `+` `-` `!` `~` `++` `--` `true` `false`

www.it-ebooks.info

binary-operator-declarator:
    type `operator` *overloadable-binary-operator* `(` *type identifier* `,` *type identifier* `)`

overloadable-binary-operator:
    `+`
    `-`
    `*`
    `/`
    `%`
    `&`
    `|`
    `^`
    `<<`
    right-shift
    `==`
    `!=`
    `>`
    `<`
    `>=`
    `<=`

conversion-operator-declarator:
    `implicit operator` *type* `(` *type identifier* `)`
    `explicit operator` *type* `(` *type identifier* `)`

operator-body:
    block
    `;`

constructor-declaration:
    attributes$_{opt}$ *constructor-modifiers*$_{opt}$ *constructor-declarator constructor-body*

constructor-modifiers:
    constructor-modifier
    constructor-modifiers constructor-modifier

constructor-modifier:
    `public`
    `protected`
    `internal`
    `private`
    `extern`

www.it-ebooks.info

constructor-declarator:
identifier (*formal-parameter-list*$_{opt}$) *constructor-initializer*$_{opt}$

constructor-initializer:
: base (*argument-list*$_{opt}$)
: this (*argument-list*$_{opt}$)

constructor-body:
block
;

static-constructor-declaration:
attributes$_{opt}$ *static-constructor-modifiers identifier* ( ) *static-constructor-body*

static-constructor-modifiers:
extern$_{opt}$ static
static extern$_{opt}$

static-constructor-body:
block
;

destructor-declaration:
attributes$_{opt}$ extern$_{opt}$ ~ *identifier* ( ) *destructor-body*

destructor-body:
block
;

B.2.8 Structs

struct-declaration:
attributes$_{opt}$ *struct-modifiers*$_{opt}$ partial$_{opt}$ struct *identifier type-parameter-list*$_{opt}$
struct-interfaces$_{opt}$ *type-parameter-constraints-clauses*$_{opt}$ *struct-body* ;$_{opt}$

struct-modifiers:
struct-modifier
struct-modifiers struct-modifier

struct-modifier:
new
public
protected
internal
private

www.it-ebooks.info

struct-interfaces:
    : *interface-type-list*

struct-body:
    { *struct-member-declarations*$_{opt}$ }

struct-member-declarations:
    struct-member-declaration
    struct-member-declarations struct-member-declaration

struct-member-declaration:
    constant-declaration
    field-declaration
    method-declaration
    property-declaration
    event-declaration
    indexer-declaration
    operator-declaration
    constructor-declaration
    static-constructor-declaration
    type-declaration

B.2.9 Arrays

array-type:
    non-array-type rank-specifiers

non-array-type:
    type

rank-specifiers:
    rank-specifier
    rank-specifiers rank-specifier

rank-specifier:
    [*dim-separators*$_{opt}$]

dim-separators:
    ,
    dim-separators ,

array-initializer:
    { *variable-initializer-list*$_{opt}$ }
    { *variable-initializer-list* , }

www.it-ebooks.info

variable-initializer-list:
    variable-initializer
    variable-initializer-list `,` *variable-initializer*

variable-initializer:
    expression
    array-initializer

B.2.10 Interfaces

interface-declaration:
    attributes$_{opt}$ *interface-modifiers*$_{opt}$ `partial`$_{opt}$ `interface`
        identifier *variant-type-parameter-list*$_{opt}$ *interface-base*$_{opt}$
            type-parameter-constraints-clauses$_{opt}$ *interface-body* `;`$_{opt}$

interface-modifiers:
    interface-modifier
    interface-modifiers *interface-modifier*

interface-modifier:
    `new`
    `public`
    `protected`
    `internal`
    `private`

variant-type-parameter-list:
    `<` *variant-type-parameters* `>`

variant-type-parameters:
    attributes$_{opt}$ *variance-annotation*$_{opt}$ *type-parameter*
    variant-type-parameters `,` *attributes*$_{opt}$ *variance-annotation*$_{opt}$ *type-parameter*

variance-annotation:
    `in`
    `out`

interface-base:
    `:` *interface-type-list*

interface-body:
    `{` *interface-member-declarations*$_{opt}$ `}`

www.it-ebooks.info

interface-member-declarations:
interface-member-declaration
interface-member-declarations *interface-member-declaration*

interface-member-declaration:
interface-method-declaration
interface-property-declaration
interface-event-declaration
interface-indexer-declaration

interface-method-declaration:
attributes$_{opt}$ `new`$_{opt}$ *return-type* *identifier* *type-parameter-list*
`(` *formal-parameter-list*$_{opt}$ `)` *type-parameter-constraints-clauses*$_{opt}$ `;`

interface-property-declaration:
attributes$_{opt}$ `new`$_{opt}$ *type* *identifier* `{` *interface-accessors* `}`

interface-accessors:
attributes$_{opt}$ `get` `;`
attributes$_{opt}$ `set` `;`
attributes$_{opt}$ `get` `;` *attributes*$_{opt}$ `set` `;`
attributes$_{opt}$ `set` `;` *attributes*$_{opt}$ `get` `;`

interface-event-declaration:
attributes$_{opt}$ `new`$_{opt}$ `event` `type` *identifier* `;`

interface-indexer-declaration:
attributes$_{opt}$ `new`$_{opt}$ *type* `this` `[` *formal-parameter-list* `]` `{` *interface-accessors* `}`

B.2.11 Enums

enum-declaration:
attributes$_{opt}$ *enum-modifiers*$_{opt}$ `enum` *identifier* *enum-base*$_{opt}$ *enum-body* `;`$_{opt}$

enum-base:
`:` *integral-type*

enum-body:
`{` *enum-member-declarations*$_{opt}$ `}`
`{` *enum-member-declarations* `,` `}`

enum-modifiers:
enum-modifier
enum-modifiers *enum-modifier*

www.it-ebooks.info

enum-modifier:
    `new`
    `public`
    `protected`
    `internal`
    `private`

enum-member-declarations:
    enum-member-declaration
    enum-member-declarations `,` *enum-member-declaration*

enum-member-declaration:
    attributes$_{opt}$ *identifier*
    attributes$_{opt}$ *identifier* `=` *constant-expression*

B.2.12 Delegates

delegate-declaration:
    attributes$_{opt}$ *delegate-modifiers*$_{opt}$ `delegate` *return-type*
        identifier variant-type-parameter-list$_{opt}$
            `(` *formal-parameter-list*$_{opt}$ `)` *type-parameter-constraints-clauses*$_{opt}$ `;`

delegate-modifiers:
    delegate-modifier
    delegate-modifiers delegate-modifier

delegate-modifier:
    `new`
    `public`
    `protected`
    `internal`
    `private`

B.2.13 Attributes

global-attributes:
    global-attribute-sections

global-attribute-sections:
    global-attribute-section
    global-attribute-sections global-attribute-section

global-attribute-section:
    `[` *global-attribute-target-specifier attribute-list* `]`
    `[` *global-attribute-target-specifier attribute-list* `,` `]`

www.it-ebooks.info

global-attribute-target-specifier:
    global-attribute-target `:`

global-attribute-target:
    `assembly`
    `module`

attributes:
    attribute-sections

attribute-sections:
    attribute-section
    attribute-sections *attribute-section*

attribute-section:
    `[` *attribute-target-specifier*$_{opt}$ *attribute-list* `]`
    `[` *attribute-target-specifier*$_{opt}$ *attribute-list* `,` `]`

attribute-target-specifier:
    attribute-target `:`

attribute-target:
    `field`
    `event`
    `method`
    `param`
    `property`
    `return`
    `type`

attribute-list:
    attribute
    attribute-list `,` *attribute*

attribute:
    attribute-name *attribute-arguments*$_{opt}$

attribute-name:
    type-name

attribute-arguments:
    `(` *positional-argument-list*$_{opt}$ `)`
    `(` *positional-argument-list* `,` *named-argument-list* `)`
    `(` *named-argument-list* `)`

www.it-ebooks.info

positional-argument-list:
    positional-argument
    positional-argument-list , *positional-argument*

positional-argument:
    argument-name$_{opt}$ *attribute-argument-expression*

named-argument-list:
    named-argument
    named-argument-list , *named-argument*

named-argument:
    identifier = *attribute-argument-expression*

attribute-argument-expression:
    expression

B.3 Grammar Extensions for Unsafe Code

class-modifier:
    ...
    `unsafe`

struct-modifier:
    ...
    `unsafe`

interface-modifier:
    ...
    `unsafe`

delegate-modifier:
    ...
    `unsafe`

field-modifier:
    ...
    `unsafe`

method-modifier:
    ...
    `unsafe`

www.it-ebooks.info

property-modifier:
    ...
    `unsafe`

event-modifier:
    ...
    `unsafe`

indexer-modifier:
    ...
    `unsafe`

operator-modifier:
    ...
    `unsafe`

constructor-modifier:
    ...
    `unsafe`

destructor-declaration:
    attributes$_{opt}$ `extern`$_{opt}$ `unsafe`$_{opt}$ `~` *identifier* `(` `)` *destructor-body*
    attributes$_{opt}$ `unsafe`$_{opt}$ `extern`$_{opt}$ `~` *identifier* `(` `)` *destructor-body*

static-constructor-modifiers:
    `extern`$_{opt}$ `unsafe`$_{opt}$ `static`
    `unsafe`$_{opt}$ `extern`$_{opt}$ `static`
    `extern`$_{opt}$ `static` `unsafe`$_{opt}$
    `unsafe`$_{opt}$ `static` `extern`$_{opt}$
    `static` `extern`$_{opt}$ `unsafe`$_{opt}$
    `static` `unsafe`$_{opt}$ `extern`$_{opt}$

embedded-statement:
    ...
    unsafe-statement
    fixed-statement

unsafe-statement:
    `unsafe` *block*

type:
    ...
    pointer-type

www.it-ebooks.info

pointer-type:
    unmanaged-type `*`
    `void` `*`

unmanaged-type:
    type

primary-no-array-creation-expression:
    ...
    pointer-member-access
    pointer-element-access
    sizeof-expression

unary-expression:
    ...
    pointer-indirection-expression
    addressof-expression

pointer-indirection-expression:
    `*` *unary-expression*

pointer-member-access:
    primary-expression `->` *identifier* *type-argument-list*$_{opt}$

pointer-element-access:
    primary-no-array-creation-expression `[` *expression* `]`

addressof-expression:
    `&` *unary-expression*

sizeof-expression:
    `sizeof` `(` *unmanaged-type* `)`

fixed-statement:
    `fixed` `(` *pointer-type* *fixed-pointer-declarators* `)` *embedded-statement*

fixed-pointer-declarators:
    fixed-pointer-declarator
    fixed-pointer-declarators `,` *fixed-pointer-declarator*

fixed-pointer-declarator:
    identifier `=` *fixed-pointer-initializer*

www.it-ebooks.info

fixed-pointer-initializer:
    `&` *variable-reference*
    expression

struct-member-declaration:
    …
    fixed-size-buffer-declaration

fixed-size-buffer-declaration:
    attributes$_{opt}$ *fixed-size-buffer-modifiers*$_{opt}$ `fixed` *buffer-element-type*
        fixed-size-buffer-declarators `;`

fixed-size-buffer-modifiers:
    fixed-size-buffer-modifier
    fixed-size-buffer-modifier *fixed-size-buffer-modifiers*

fixed-size-buffer-modifier:
    `new`
    `public`
    `protected`
    `internal`
    `private`
    `unsafe`

buffer-element-type:
    type

fixed-size-buffer-declarators:
    fixed-size-buffer-declarator
    fixed-size-buffer-declarator `,` *fixed-size-buffer-declarators*

fixed-size-buffer-declarator:
    identifier `[` *constant-expression* `]`

local-variable-initializer:
    …
    stackalloc-initializer

stackalloc-initializer:
    `stackalloc` *unmanaged-type* `[` *expression* `]`

www.it-ebooks.info

C. References

IEEE. *IEEE Standard for Binary Floating-Point Arithmetic*. ANSI/IEEE Standard 754-1985. Available from http://www.ieee.org.

ISO/IEC. *C++*. ANSI/ISO/IEC 14882:1998.

Unicode Consortium. *The Unicode Standard, Version 3.0*. Addison-Wesley, Reading, Massachusetts, 2000, ISBN 0-201-616335-5.

www.it-ebooks.info

This page intentionally left blank

www.it-ebooks.info

Index

A

\a escape sequence, 81
Abstract accessors, 46, 557–558
Abstract classes
  and interfaces, 661
  overview, 468–469
Abstract events, 566
Abstract indexers, 567
Abstract methods, 35, 539–540
Access and accessibility
  array elements, 628
  containing types, 502–503
  events, 253
  indexers, 253, 300–301
  members, 23–24, 107, 496
    accessibility domains, 110–113
    constraints, 116–117
    declared accessibility, 107–109
    interface, 642–644
    pointer, 721–722
    in primary expressions, 283–288
    protected, 113–116
  nested types, 499–503
  pointer elements, 723
  primary expression elements, 298–301
  properties, 252, 555–556
Accessors
  abstract, 46, 557–558
  attribute, 695
  event, 564–565
  property, 43, 46, 547–553
Acquire semantics, 514
Acquisition in using statement, 445–446
add accessors
  attributes, 695
  events, 49, 564
Add method
  IEnumerable, 311
  List, 42
AddEventHandler method, 565
Addition operator
  described, 15
  uses, 337–340
Address-of operator, 724–725
Addresses
  fixed variables, 728–733
  pointers for, 714, 724–725
after state for enumerator objects, 593–596
Alert escape sequence, 81
Aliases
  for namespaces and types, 456–461
  qualifiers, 464–466
  uniqueness, 466
Alignment enumeration, 58–59
Alloc method, 738
Allocation, stack, 736–738
AllowMultiple parameter, 688

www.it-ebooks.info

Ambiguities
grammar, 287–288
in query expressions, 376
Ampersands (&)
for addresses, 716
in assignment operators, 389
definite assignment rules, 188–189
for logical operators, 355–359
for pointers, 724–725
in preprocessing expressions, 87
AND operators, 15
Angle brackets (<>) for type arguments, 160
Anonymous functions
bodies, 367
conversions, 219–221
evaluation to delegate types, 165–166, 221–222
evaluation to expression tree types, 222
implementation example, 222–226
implicit, 204
definite assignment rules, 192
delegate creation, 61
dynamic binding, 369
evaluation of, 373
expressions, 165–166, 326, 364–366
outer variables, 369–373
overloaded, 368
signatures, 365–366
Anonymous objects, 317–319
AppendFormat method, 31
Applicable function members, 271–272
Application domains, 99
ApplicationExcpetion class, 681
Applications, 4
startup, 99–100
termination, 100–101
Apply method, 60
Arguments, 28. *See also* Parameters
command-line, 99
for function members, 254–259
type, 161–162
type inference, 259–270
Arithmetic operators, 331–332
addition, 337–340
division, 334–335
multiplication, 332–333
pointer, 725–726
remainder, 336–337
shift, 344
subtraction, 340–342
ArithmeticException class, 335, 685
Arrays and array types, 625
access to, 299–300, 628
content, 13
conversions, 200
covariance, 200, 629–630
creating, 628
description, 8, 155
elements, 53, 171, 628
with foreach, 429
IList interface, 627–628
initializers, 55, 630–632
members, 106, 628
new operator for, 53, 55, 312–315
overview, 53–55
parameter, 31, 528–531
and pointers, 719–720, 730–731
rank specifiers, 625–626
syntactic grammar, 804–805
ArrayTypeMismatchException class
description, 685
type mismatch, 390–391, 629
as operator, 353–355
Assemblies, 4–5
Assignment
in classes *vs.* structs, 612
definite. *See* Definite assignment
fixed size buffers, 736
Assignment operators, 16
compound, 393–394
event, 394–395
overview, 389–390
simple, 390–393
Associativity of operators, 238–240
Asterisks (*)
assignment operators, 389
comments, 69–70, 741–742
multiplication, 332–333
pointers, 713–716, 721
transparent identifiers, 385

www.it-ebooks.info

At sign characters (@) for identifiers, 72-74
Atomicity of variable references,193
Attribute class, 62,688
Attributes, 687
  classes, 688–691, 704–705
  compilation of, 698–699
  compilation units,454
  instances, 698–699
  for interoperation,707
  overview, 61–63
  parameters for, 690-691
  partial types, 482–483
  reserved, 699–700
    AttributeUsage, 700
    Conditional, 701–705
    Obsolete, 705–706
  sections for,692
  specifications, 692–698
  syntactic grammar, 807-809
AttributeUsage attribute, 688–690,700
Automatic memory management, 132–137
Automatically implemented properties,548, 553–555

B

\b escape sequence,81
Backslash characters (\)
  for characters, 80–81
  escape sequence, 80–81
  for strings,82
Backspace escape sequence,81
Backtick character (`),83
Banker's rounding,150
Base access, 302–303
Base classes, 22, 25–26
  partial types,484
  specifications for, 472-475
  type parameter constraints,476
Base interfaces
  inheritance from, 637–638
  partial types, 484–485
Base types, 249–250
before state for enumerator objects, 593-596
Better conversions, 274–275
Better function members, 272–273
Binary operators,238
  declarations, 574–575
  in ID string format,759
  lifted, 246
  numeric promotions, 244–245
  overload resolution,243
  overloadable, 241
Binary point types,9
Bind method,56
Binding
  constituent expressions,237
  dynamic, 166, 234–237
  name, 490
  static, 234–235
  time, 235
BitArray class, 569–570
Bitwise complement operator,328
Blocks
  in declarations, 102–104
  definite assignment rules,179
  exiting, 430
  in grammar notation,66
  invariant meaning in, 281–283
  in methods,544
  reachability of,401
  in statements, 402–404
  for unsafe code,710
Bodies
  classes, 481
  interfaces, 638
  methods, 32–33,544
  struct, 609
Boneheaded exceptions,686
bool type, 8–9, 150–151
Boolean values
  expressions,397
  literals,76
  operators
    conditional logical,359
    equality,348
    logical, 357
  in struct example, 622-623
Boss class,47
Bound types,162
Box class,539

www.it-ebooks.info

Boxed instances, invocations on, 278
Boxing, 12, 155–156
  in classes *vs.* structs, 613–616
  conversions, 156–158, 201
break statement
  definite assignment rules, 182
  example, 19
  for for statements, 423
  overview, 431
  for switch, 416–417
  for while, 420
  yield break, 449–452, 594–595
Brittle base class syndrome, 35, 292
Brittle derived class syndrome, 292, 297
Buffers, fixed-size
  declarations, 733–735
  definite assignment, 736
  in expressions, 735–736
Bugs. *See* Unsafe code
Button class, 549, 561
byte type, 10

C

<c> tag, 744
Cache class, 444
Callable entities, 671
Candidate user-defined operators, 243
Captured outer variables, 369–370
Carets (^)
  in assignment operators, 389
  for logical operators, 355–357
Carriage-return characters
  escape sequence, 81
  as line terminators, 68–69
Case labels, 415–419
Cast expressions, 330–331
cast operator *vs.* as operator, 355
catch blocks
  definite assignment rules, 183–185
  for exceptions, 684–685
  throw statements, 436–437
  try statements, 438–443
char type, 146
Character literals, 80–81
Characters, 9
checked statement
  definite assignment rules, 179
  example, 20
  overview, 443
  in primary expressions, 322–325
Chooser class, 259–260
Classes
  accessibility, 23
  attribute, 688–691, 704–705
  base, 25–26, 472–475
  bodies, 481
  constants for, 506–508
  constructors for, 42–43
    instance, 579–586
    static, 586–589
  declarations, 467
    base specifications, 472–475
    bodies, 481
    modifiers, 467–471
    partial type, 471
    type parameter constraints, 475–481
    type parameters, 471–472
  defined, 467
  destructors for, 50, 589–591
  events in, 47–49
    accessors, 564–565
    declaration, 559–562
    field-like, 562–564
    instance and static, 565
  fields in, 26–27
    declarations, 509–510
    initializing, 515–516
    read-only, 511–513
    static and instance, 510–511
    variable initializers, 516–519
    volatile, 514–515
  function members in, 40–50
  indexers in, 46–47, 566–571
  instance variables in, 170–171
  interface implementation by, 57
  iterators. *See* Iterators

www.it-ebooks.info

members in, 22–23, 106, 490–492
access modifiers for,496
constituent types for,496
constructed types, 493–494
inheritance of, 494–496
instance types, 492
nested types for, 498–504
new modifier for,496
reserved names for, 504–506
static and instance, 496–498
methods in, 28–40
abstract, 539–540
bodies, 544
declaration, 520–522
extension, 541–543
external, 539–540
parameters, 522–531
partial, 541
sealed, 537–538
static and instance,531
virtual, 532–534
operators in, 49–50
binary, 574–575
conversion, 575–578
declaration, 571–573
unary, 573–574
overview, 21–22
partial types. *See* Partial types
in program structure, 4–5
properties in, 43–46
accessibility, 555–556
accessors for, 547–553
automatically implemented, 553–555
declarations, 545–546
static and instance,546
vs. structs, 610–619
syntactic grammar, 794–803
type parameters, 24–25
types, 6–13, 153–154
Classifications, expression, 231–234
Click events, 562–563
Closed types, 162
CLS (Common Language Specification),9
<code> tag, 744
Collections
for foreach, 425
initializers, 310–312
Colons (:)
alias qualifiers, 464–465
grammar productions, 66
interface identifiers, 637
ternary operators, 191, 361–362
type parameter constraints,476
Color class, 27,512
Color enumeration, 58, 664–666
Color struct, 286
COM, interoperation with,707
Combining delegates, 340,675
Command-line arguments, 99
Commas (,)
arrays, 54
attributes, 692
collection initializers, 310
ID string format,755
interface identifiers, 637
method parameter lists,522
object initializers, 307
Comments, 741
documentation file processing, 754–759
example, 760–766
lexical grammar, 69–70,768
overview, 741–743
tags, 743–753
XML for, 741–742, 762-765
Commit method, 92
Common Language Specification (CLS),9
Common types for type inference,270
CompareExchange method, 600
Comparison operators, 49
booleans, 348
decimal numbers, 348
delegates, 351–352
enumerations, 348
floating point numbers, 346–347
integers, 346
overview, 344–345
pointers, 726
reference types, 349–351
strings, 351

www.it-ebooks.info

Compatibility of delegates and methods, 676
Compilation
  attributes, 698–699
  binding, 235
  dynamic overload resolution checking, 275–276
  just-in-time, 5
Compilation directives, 90–93
Compilation symbols, 87
Compilation unit productions, 67
Compilation units, 65, 453–454
Compile-time type of instances, 35, 532
Complement operator, 328
Component-oriented programming, 1–2
Compound assignment
  operator, 389
  process, 393–394
Concatenation, string, 339
Conditional attribute, 701–705
Conditional classes, 704–705
Conditional compilation directives, 90–93
Conditional compilation symbols, 87
Conditional logical operators, 15, 358–360
Conditional methods, 701–703
Conditional operator, 15, 361–363
Console class, 31, 552–553
Constant class, 35–36
Constants, 41
  declarations, 411–412, 506–508
  enums for. *See* Enumerations and enum types
  expressions, 203, 395–397
  static fields for, 512–513
  versioning of, 512–513
Constituent expressions, 237
Constituent types, 496
Constraints
  accessibility, 116–117
  constructed types, 162–164
  partial types, 483–484
  type parameters, 475–481
Constructed types, 160–161
  bound and unbound, 162
  constraints, 162–164
  members, 493–494
  open and closed, 162
  type arguments, 161
Constructors, 41
  for classes, 42–43
  for classes *vs.* structs, 617–618
  default, 141–142, 584
  in ID string format, 757
  instance. *See* Instance constructors
  invocation, 254
  static, 42, 586–589
Contact class, 311
Contexts
  for attributes, 694–696
  unsafe, 710–713
Contextual keywords, 75
continue statement
  definite assignment rules, 182
  for do, 421
  example, 19
  for for statements, 423
  overview, 432
  for while, 420
Contracts, interfaces as, 633
Contravariant type parameters, 635
Control class, 564–565
Control-Z character, 68
Conversions, 195
  anonymous functions, 165–166, 219–226, 365–366
  boxing, 156–158, 201
  constant expression, 203
  dynamic, 202, 210
  enumerations, 198, 207
  explicit, 204–213
  expressions, 330–331
  function members, 274–275
  identity, 196–197
  implicit, 195–204
    standard, 213
    user-defined, 217–219
  method groups, 226–229
  null literal, 199
  nullable, 198–199, 207–208, 360–361
  numeric, 197, 205–207
  as operator for, 353–355

www.it-ebooks.info

operators, 575–578, 759
for pointers, 717–720
reference, 199–201, 208–210
standard, 213–214
type parameters, 203–204, 211-212
unboxing, 158–160, 210
user-defined. *See* User-defined conversions
variance, 636
Convert class, 207
Copy method, 739
Counter class, 552
Counter struct, 614
CountPrimes class, 570
Covariance
array, 200, 629-630
type parameters, 635
cref attribute, 742
Critical execution points, 137
.cs extension, 3
Curly braces ({})
arrays, 55
collection initializers, 310
grammar notation, 66
object initializers, 307
Currency type, 150
Current property, 595
Customer class, 488–489

D

Database structure example
boolean type, 622–623
integer type, 619–621
DBBool struct, 622–623
DBInt struct, 619–621
Decimal numbers and type, 9-10
addition, 338–339
comparison operators, 348
division, 335
multiplication, 333
negation, 327
remainder operator, 337
subtraction, 341
working with, 149–150
decimal128 type, 149
Declaration directives, 88–89
Declaration space, 101
Declaration statements, 407–412
Declarations
classes, 467
base specifications, 472–475
bodies, 481
modifiers, 467–471
partial type, 471
type parameter constraints, 475–481
type parameters, 471–472
constants, 411–412, 506–508
definite assignment rules, 180
delegates, 672–675
enums, 59, 663–664
events, 559–562
fields, 509–510
fixed-size buffers, 733–735
indexer, 566–571
instance constructors, 579–580
interfaces, 633–638
methods, 520–522
namespaces, 103, 454–456
operators, 571–573
order, 6, 103
overview, 101–104
parameters, 522–525
pointers, 714
properties, 545–546
property accessors, 547
static constructors, 586–589
struct members, 609
structs, 608–609
types, 10, 464
variables, 175, 407–411
Declared accessibility
nested types, 499–500
overview, 107–109
Decrement operators
pointers, 725
postfix, 303–305
prefix, 328–330
default expressions, 142

www.it-ebooks.info

Defaults
constructors, 141–142, 584
switch statement labels, 415–416
values, 141, 175–176
classes *vs.* structs, 612–613
expressions, 325
#define directive, 87, 89
Defining partial method declarations, 486
Definite assignment, 33, 169, 176–177
fixed size buffers, 736
initially assigned variables, 177
initially unassigned variables, 177
rules for, 178–192
Degenerate query expressions, 379–380
Delegate class, 671
Delegates and delegate type, 671–672
combining, 340, 675
compatible, 676
contents, 13
conversions, 165–166, 221–222
declarations, 672–675
description, 8, 11, 155
equality, 351–352
instantiation, 676–677
invocations, 298, 677–680
members of, 107
new operator for, 315–317
overview, 60–61
removing, 342
syntactic grammar, 807
Delimited comments, 69–70, 741–742
Dependence
on base classes, 473–474
in structures, 611
type inference, 263
Depends on relationships, 473–474, 611
Derived classes, 22, 25–26
Destructors
for classes, 50, 589–591
for classes *vs.* structs, 619
exceptions for, 685
garbage collection, 132–137
in ID string format, 757
member names reserved for, 506
members, 23
Diagnostic directives, 93–94
Digit struct, 578
Dimensions, array, 11, 54, 625, 631–632
Direct base classes, 472–473
Directives
preprocessing. *See* Preprocessing directives
using. *See* Using directives
Directly depends on relationships, 473–474, 611
Disposal in using statement, 446
Dispose method, 591
for enumerator objects, 596, 604–605
for resources, 445–446
Divide method, 30
DivideByZeroException class, 333–334, 683, 685
Division operator, 334–335
DllImport attribute, 541
DLLs (Dynamic Link Libraries), 541
do statement
definite assignment rules, 181–182
example, 18
overview, 421
Documentation comments, 741
documentation files for, 741, 754
ID string examples, 755–759
ID string format, 754–756
example, 760–766
overview, 741–743
tags for, 743–753
XML files for, 741–742, 762–765
Documentation generators, 741
Documentation viewers, 741
Domains
accessibility, 110–113
application, 99
Double quotes (")
characters, 80
strings, 80
double type, 9–10, 146–149
DoubleToInt64Bits method, 334

www.it-ebooks.info

Dynamic binding
anonymous functions, 369
overview, 234–237
Dynamic Link Libraries (DLLs), 541
Dynamic memory allocation, 738–740
Dynamic overload resolution, 275–276
dynamic type, 154
conversions, 202, 210
identity conversions, 197
overview, 166–167

E

ECMA-334 standard, 1
EditBox class, 56–57
Effective base classes, 480
Effective interface sets, 480
Elements
array, 53, 171, 628
foreach, 425–427
pointer, 723
primary expression, 298–301
#elif directive, 87–88, 91
Ellipse class, 539
#else directive, 87, 90–93
Embedded statements and expressions
general rules, 186–187
in grammar notation, 66
Empty statements, 404–406
Encompassed types, 216
Encompassing types, 216
End-of-file markers, 68
End points, 400–402
#endif directive, 91
#endregion directive, 94
Entity class, 33–34
Entry class, 5
Entry points, 99
Enumerable interfaces, 592
Enumerable objects for iterators, 596–597
Enumerations and enum types
addition of, 339
comparison operators, 348
conversions
explicit, 207
implicit, 198
declarations, 663–664
description, 8, 11, 663, 668
logical operators, 356–357
members, 106, 665–668
modifiers, 664–665
overview, 58–59
subtraction of, 341
syntactic grammar, 806–807
types for, 151
values and operations, 668–669
Enumerator interfaces, 592
Enumerator objects for iterators, 593–596
Enumerator types for foreach, 425-426
Equal signs (=)
assignment operators, 389
comparisons, 345
operator ==, 49–50
pointers, 726
preprocessing expressions, 87
Equality operators, 15
boolean values, 348
delegates, 351–352
lifted, 246–247
and null, 352
reference types, 349–351
strings, 351
Equals method
on anonymous types, 319
DBBool, 623
DBInt, 621
List, 42
with NaN values, 347
Point, 761
#error directive, 94
Error property, 553
Error strings in ID string format, 754
Escape sequences
characters, 81
lexical grammar, 769
strings, 81
unicode character, 71–72
Evaluate method, 37
Evaluation of user-defined conversions, 215–216
Event handlers, 48, 559, 562

www.it-ebooks.info

Events, 4
access to, 253
accessors, 564–565
assignment operator, 394–395
declarations, 559–562
example, 42
field-like, 562–564
in ID string format, 754, 758–759
instance and static, 565
interface, 642
member names reserved for, 506
overview, 47–49
Exact parameter type inferences, 264
<example> tag, 745
Exception class, 436, 438, 682–684
Exception propagation, 437
<exception> tag, 745
Exception variables, 438
Exceptions
causes, 683
classes for, 685–686
for delegates, 677
handling, 1, 684–685
overview, 681–682
throwing, 436–437
try statement for, 438–443
Exclamation points (!)
comparisons, 345
definite assignment rules, 190
logical negation, 327
operator !=, 49
pointers, 726
preprocessing expressions, 87
Execution
instance constructors, 582–584
order of, 137–138
Exiting blocks, 430
Exogenous exceptions, 686
Expanded form function members, 272
Explicit base interfaces, 637
Explicit conversions, 204–205
dynamic, 210
enumerations, 207
nullable types, 207–208
numeric, 205–207
reference, 208–210
standard, 214
type parameters, 211–212
unboxing, 210
user-defined, 213, 218–219
Explicit interface member implementations, 57, 647–650
explicit keyword, 576–578
Explicit parameter type inferences, 264
Expression class, 35–37
Expression statements, 17, 179, 412-413
Expressions, 231
anonymous function. *See* Anonymous functions
binding, 234–237
boolean, 397
cast, 330–331
classifications, 231–234
constant, 203, 395–397
constituent, 237
definite assignment rules, 186–191
dynamic, 166
fixed-size buffers in, 735–736
function members
argument lists, 254–259
categories, 250–254
invocation, 276–278
overload resolution, 270–275
type inference, 259–270
member lookup, 247–250
operators for, 238
arithmetic. *See* Arithmetic operators
assignment, 389–395
logical, 355–357
numeric promotions, 244–246
overloading, 240–243
precedence and associativity, 238-240
relational, 345
shift, 343–344
unary, 326–331
overview, 13–16
pointers in, 720–727
preprocessing, 87–88
primary. *See* Primary expressions

www.it-ebooks.info

query, 373–375
ambiguities in, 376
patterns, 387–389
translations in, 376–387
syntactic grammar, 779–788
tree types, 165–166, 222
values of, 233
Extensible Markup Language (XML), 741–742, 762–765
Extension methods
example, 541–543
invocation, 293–297
Extensions class, 542
extern aliases, 456–457
External constructors, 580, 586
External destructors, 589
External events, 560
External indexers, 569
External methods, 539–540
External operators, 572
External properties, 546

F

\f escape sequence, 81
False value, 76
Fatal exceptions, 686
Field-like events, 562–564
Fields, 4
declarations, 509–510
example, 41
in ID string format, 754, 756-757
initializing, 515–516, 616–617
instance, 26–27, 510–511
overview, 26–27
read-only, 27–28, 511-513
static, 510–511
variable initializers, 516–519
volatile, 514–515
Fill method, 629
Filters, 442
Finalize method, 591
Finalizers, 50
finally blocks
definite assignment rules, 184–185
for exceptions, 684
execution, 682
with goto, 434
with try, 438–443
Fixed-size buffers
declarations, 733–735
definite assignment, 736
in expressions, 735–736
fixed statement, 716, 728–733
Fixed variables, 716–717
Fixing type inferences, 266–267
float type, 9–10, 146–149
Floating point numbers
addition, 338
comparison operators, 346–347
division, 334–335
multiplication, 332
NaN payload, 333–334
negation, 327
remainder operator, 336
subtraction, 340–341
types, 9–10, 146–149
for statement
definite assignment rules, 182
example, 19
overview, 422–423
foreach statement
definite assignment rules, 185
example, 19
overview, 423–429
Form feed escape sequence, 81
Forward declarations, 6
Fragmentation, heap, 729
Free method, 739
from clauses, 375, 379–387
FromTo method, 599-600
Fully qualified names
described, 131
interface members, 645
nested types, 499
Function members
argument lists, 254–259
in classes, 40–50
dynamic overload resolution checking, 275–276
overload resolution, 270–275
overview, 250–254
type inference, 259–270

www.it-ebooks.info

Function pointers, 671
Functional notation, 241
Functions, anonymous. *See* Anonymous functions

G

Garbage collection, 1
  at application termination, 101
  for destructors, 50
  in memory management, 132–137, 176
  and pointers, 713
  for variables, 716
GC class, 133, 136
Generic classes and types, 25, 139
  anonymous objects, 318
  boxing, 156, 613
  constraints, 162, 475–477, 483
  declarations, 467, 473
  delegates, 220
  instance type, 492
  interfaces, 650–651
  member lookup, 247
  methods, 521, 532, 652–653
  nested, 247, 503
  overloading, 275
  overriding, 536
  query expression patterns, 387
  signatures, 28
  static fields, 26
  type inferences, 259–261, 267
  unbound, 160
Generic interface, 627–628
get accessors
  for attributes, 695
  defined, 45
  description, 557
  working with, 547–553
GetEnumerator method
  for foreach, 425
  for iterators, 596–603
GetEventHandler method, 565
GetHashCode method
  on anonymous types, 319
  comparisons, 347
  DBBool, 623
  DBInt, 621
GetHourlyRate method, 38
GetInvocationList method, 677
GetNextSerialNo method, 34
GetProcessHeap method, 739
Global declaration space, 101
Global namespace, 105
goto statement
  definite assignment rules, 182
  example, 19
  for switch, 416–417, 419
  working with, 433–434
Governing types of switch statements, 415, 418
Grammars, 65
  ambiguities, 287–288
  lexical. *See* Lexical grammar
  notation, 65–67
  syntactic. *See* Syntactic grammar
  for unsafe code, 809–812
Greater than signs (>)
  assignment operators, 389
  comparisons, 345
  pointers, 716, 721–722, 726
  shift operators, 343–344
Grid class, 570–571
group clauses, 375, 378, 385

H

Handlers, event, 48, 559, 562
HasValue property, 152
Heap, 7
  accessing functions of, 738–740
  fragmentation, 729
HeapAlloc method, 739
HeapFree method, 739
HeapReAlloc method, 740
HeapSize method, 740
Hello, World program, 3

www.it-ebooks.info

Hello class, 93
HelpAttribute class, 61–62, 690
HelpStringAttribute class, 697
Hexadecimal escape sequences
  for characters, 80
  for strings, 83
Hiding
  inherited members, 102, 125–127, 495
  in multiple-inheritance interfaces, 644
  in nesting, 124–127, 500
  properties, 550
  in scope, 120
Hindley-Milner-style algorithms, 261
Horizontal tab escape sequence, 81

I

IBase interface, 644, 655, 660
ICloneable interface, 645–646, 649, 654
IComboBox interface, 56, 638
IComparable interface, 646
IControl interface, 56–57, 638
  implementations, 646
  inheritance, 657–659
  mapping, 654–656
  member implementations, 650
  member names, 645
  reimplementations, 659–660
ICounter interface, 643
ICounter struct, 615
ID string format
  for documentation files, 754–756
  examples, 755–759
IDataBound interface, 56–57
Identical simple names and type names, 286–287
Identifiers
  interface, 634
  lexical grammar, 769–770
  rules for, 72–74
Identity conversions, 196–197
IDerived interface, 655
IDictionary interface, 648
IDisposable interface, 136, 428, 445–447, 591, 648
IDouble interface, 644
IEnumerable interface, 311, 427–428, 596–597
IEnumerator interface, 592
#if directive, 87–88, 90–93
if statement
  definite assignment rules, 180
  example, 18
  working with, 413–414
IForm interface, 655
IInteger interface, 643–644
IL (Intermediate Language) instructions, 5
IList interface, 627–628, 643, 647
IListBox interface, 56, 638, 656
IListCounter interface, 643
IMethods interface, 659–661
Implementing partial method declarations, 486
Implicit conversions, 195–196
  anonymous functions and method groups, 204
  boxing, 201
  constant expression, 203
  dynamic, 202
  enumerations, 198
  identity, 196
  null literal, 199
  nullable, 198–199
  numeric, 197
  operator for, 575–578
  standard, 213
  type parameters, 203–204
  user-defined, 204, 217
implicit keyword, 575–578
Implicitly typed array creation expressions, 313
Implicitly typed iteration variables, 423, 427
Implicitly typed local variable declarations, 408–409
Importing types, 461–463
In-line methods, 61
In property, 553
Inaccessible members, 107

www.it-ebooks.info

<include> tag, 742, 745–746
Increment operators
  for pointers, 725
  postfix, 303–305
  prefix, 328–330
IndexerName Attribute, 707
Indexers
  access to, 253, 300–301
  declarations, 566–571
  example, 42
  in ID string format, 758
  interface, 642
  member names reserved for, 506
  overview, 46–47
  signatures in, 119
IndexOf method, 39–40
IndexOutOfRangeException class, 300, 685
Indices, array, 53
Indirection, pointer, 716, 721
Inference, type, 259–270
Infinity values, 147–148
Inheritance, 22
  from base interfaces, 637–638
  in classes, 25–26, 105, 494–496
  in classes *vs.* structs, 612
  hiding through, 102, 125–127, 495
  interface, 640, 657–659
  parameters, 689
  properties, 550
Initializers
  array, 55, 630–632
  field, 515–516, 616–617
  in for statements, 422
  instance constructors, 580–581
  stack allocation, 736–738
  variables, 516–519, 581
Initially assigned variables, 169, 177
Initially unassigned variables, 169, 177
Inlining process, 552
InnerException property, 683
Input production, 67
Input-safe types, 636
Input types in type inference, 263
Input-unsafe types, 636
Instance constructors
  declarations, 579–580
  default, 584
  description, 42
  execution, 582–584
  initializers, 580–581
  invocation, 254
  optional parameters, 585–586
  private, 584–585
Instance events, 565
Instance fields
  class, 510–511
  example, 26–27
  initialization, 515–516, 519
  read-only, 511–513
Instance members
  class, 496–498
  description, 22
  protected access for, 113–116
Instance methods, 28, 33–34, 531
Instance properties, 546
Instance types, 492
Instance variables, 170–171, 510–511
Instances, 21–22
  attribute, 698–699
  type, 153
Instantiation
  delegates, 676–677
  local variables, 370–373
int type, 9–10
Int64BitsToDouble method, 334
Integers
  addition, 338
  comparison operators, 346
  division, 334
  literals, 76–78
  logical operators, 356
  multiplication, 332
  negation, 327
  remainder, 336
  in struct example, 619–621
  subtraction, 340
Integral types, 9–10, 145–146
interface keyword, 634
Interface sets, 480

www.it-ebooks.info

Interfaces, 4, 633
base, 637–638
bodies, 638
declarations, 633–638
enumerable, 592
enumerator, 592
generic, 650–651
implementations, 645–647
abstract classes, 661
base classes, 475
explicit member, 647-650
generic methods, 652–653
inheritance, 657–659
mapping, 653–656
reimplementation, 659–660
uniqueness, 650–652
inheritance from, 637–638
members, 106, 639–640
access to, 642–644
events, 642
fully qualified names, 645
indexers, 642
methods, 640–641
properties, 641–642
modifiers, 634
overview, 56–57
partial types, 484–485
struct, 609
syntactic grammar, 805-806
types, 8, 11–13,155
variant type parameter lists, 635-637
Intermediate Language (IL) instructions,5
Internal accessibility, 23,107
Interning, 84
Interoperation attributes, 707
IntToString method, 737–738
IntVector class,574
InvalidCastException class, 159, 210, 355,685
InvalidOperationException class, 152, 600
Invariant meaning in blocks, 281–283
Invariant type parameters, 635
Invocable members, 247–248
Invocation
delegates, 298, 677–680
function members, 276–278
instance constructors, 254
methods, 251
operators, 254
Invocation expressions, 187, 288–298
Invocation lists, 675, 677
Invoked members, 247–248
IronPython, 236
is operator, 352–353
isFalse property,622
IsNan method, 334
isNull property
DBBool, 622
DBInt, 620
ISO/IEC 23270 standard, 1
IStringList interface, 639
isTrue property,622
Iteration statements, 420
do, 421
for, 422–423
foreach, 423–429
while, 420–421
Iteration variables in foreach, 423-424
Iterators, 592
blocks, 403
enumerable interfaces, 592
enumerable objects for, 596–597
enumerator interfaces, 592
enumerator objects for, 593–596
implementation example, 597–603
yield type, 592
ITest interface, 119-120
ITextBox interface, 56, 638, 645–647, 650,656

J

Jagged arrays, 54
JIT (Just-In-Time) compiler,5
join clauses, 380–384
Jump statements
break, 431
continue, 432
goto, 433–434
overview, 429–431
return, 435
throw, 436–437
Just-In-Time (JIT) compiler,5

www.it-ebooks.info

K

KeyValuePair struct,613
Keywords
  lexical grammar,770
  list, 74–75

L

Label class, 551–552
Label declaration space, 102–103
Labeled statements
  for goto, 433–434
  overview, 406–407
  for switch, 181, 415–419
Left-associative operators, 239
Left shift operator, 343-344
Length of arrays, 53, 625, 631-632
Less than signs (<)
  assignment operators, 389
  comparisons, 345
  pointers, 726
  shift operators, 343–344
let clauses, 380–384
Lexical grammar, 67,767
  comments, 69–70, 768
  identifiers, 769–770
  keywords, 770
  line terminators, 68–69,767
  literals, 771–773
  operators and punctuators,773
  preprocessing directives, 774–777
  tokens, 769
  unicode character escape sequences,769
  whitespace, 70–71, 769
Lexical structure, 65
  grammars, 65–67
    lexical. *See* Lexical grammar
    syntactic. *See* Syntactic grammar
  lexical analysis, 67–71
  preprocessing directives, 85–87
    conditional compilation, 87, 90–93
    declaration, 88–89
    diagnostic, 93–94
    line, 95–96
    pragma, 96–97
    preprocessing expressions, 87–88
    region, 94
  programs, 65
  tokens, 71
    identifiers, 72–74
    keywords, 74–75
    literals, 76–84
    operators, 84–85
    unicode character escape sequence, 71–72
Libraries, 4, 541
Lifted conversions, 215
Lifted operators, 246–247
#line directive, 94
#line default directive, 96
Line directives, 95–96
Line-feed characters, 69
#line hidden directive, 96
Line-separator characters, 69
Line terminators, 68–69, 767
List class, 40–50
<list> tag, 746–747
ListChanged method, 48
Lists, statement, 403–404
Literals
  boolean, 76
  character, 80–81
  in constant expressions, 395
  conversions, 199
  defined, 76
  integer, 76–78
  lexical grammar, 771–773
  null, 84
  in primary expressions, 279
  real, 78–79
  simple values, 144
  string, 81–84
Local constant declarations, 17, 411-412
Local variable declaration space, 103
Local variables
  declarations, 17, 407–411
  instantiation, 370–373
  in methods, 32–33
  scope, 124–125
  working with, 173–175

www.it-ebooks.info

lock statement
definite assignment rules, 186
example, 21
overview, 443–445
Logical operators
AND, 15
for boolean values, 357
conditional, 358–360
for enumerations, 356–357
for integers, 356
negation, 327–328
OR, 15
overview, 356–357
shift, 344
XOR, 15
LoginDialog class, 561
long type, 9–10
Lookup, member, 247–250
Lower-bound type inferences, 264–265
lvalues, 193

M

Main method
for startup, 99–100
for static constructors, 587–588
Mappings
interface, 653–656
pointers and integers, 719
Math class, 334
Members, 4, 22–23, 105
access to, 23, 107, 496
accessibility domains, 110–113
constraints, 116–117
declared accessibility, 107–109
interface, 642–644
pointer, 721–722
in primary expressions, 283–288
protected, 113–116
accessibility of, 23–24
array, 106, 628
class, 106, 490–492
access modifiers for, 496
constituent types, 496
constructed types, 493–494
inheritance of, 494–496
instance types, 492
nested types, 498–504
new modifier for, 496
reserved names for, 504–506
static and instance, 496–498
delegate, 107
enumeration, 106, 665–668
function. *See* Function members
inherited, 102, 105, 125–127, 494–496
interface, 106, 639–640
access to, 642–644
events, 642
explicit implementations, 57, 647–650
fully qualified names, 645
indexers, 642
methods, 640–641
properties, 641–642
lookup, 247–250
namespaces, 105, 463–464
partial types, 485
pointer, 721–722
struct, 105–106, 609
Memory
automatic management of, 132–137, 176
dynamic allocation of, 738–740
Memory class, 738–740
Memory leaks from events, 561
Message property, 683
Metadata, 5
Method group conversions
implicit, 204
overview, 226–229
type inference, 269
Methods, 4, 28
abstract, 35, 539–540
bodies, 32–33, 544
conditional, 701–703
declarations, 520–522
extension, 541–543
external, 539–540
in ID string format, 754, 757–758
instance, 28, 33–34, 531
interface, 640–641
invocations, 251, 288–298
in List, 42

www.it-ebooks.info

Methods *(continued)*
overloading, 38–40
overriding, 35, 535–537
parameters, 29–32
arrays, 528–531
declarations, 522–525
output, 526–527
reference, 525–526
value, 525
partial, 486–490, 541
sealed, 537–538
static, 28, 33–34, 531
virtual, 35–38, 532–534
Minus (-) operator, 327
Minus signs (-)
assignment operators, 389
decrement operator, 303–305, 328–330
pointers, 716, 721–722, 725
subtraction, 340–342
Modifiers
class, 467–471
enums, 664–665
interface, 634
partial types, 483
struct, 609
Modulo operator, 336–337
Most derived method implementation, 532–533
Most encompassing types, 216
Most specific operators, 215
Move method, 760–761
Moveable variables
described, 716–717
fixed addresses for, 728–733
MoveNext method, 426
enumerator objects, 451, 593–595
Stack, 599
Tree, 603–604
Multi-dimensional arrays, 11, 54, 625, 631–632
Multi-use attribute classes, 688
Multiple inheritance, 56–57, 644
Multiple statements, 402–403
Multiplication operator, 15, 332–333
Multiplicative operators, 15
Multiplier class, 60–61
Multiply method, 60
Mutual-exclusion locks, 443–445

N

\n escape sequence, 81
Named constants. *See* Enumerations and enum types
Named parameters, 690–691
Names
anonymous types, 318–319
binding, 490
fully qualified, 131
interface members, 645
nested types, 499
hiding, 124–127
methods, 521
reserved, 504–506
simple
in primary expressions, 279–283
and type names, 286–287
variables, 170
namespace keyword, 454
Namespaces, 3–4, 453
aliases, 456–461, 464–466
compilation units, 453–454
declarations, 103, 454–456
using directives in, 457–463
fully qualified names in, 131
in ID string format, 754
members, 105, 463–464
overview, 127–130
purpose, 104
syntactic grammar, 793–794
type declarations, 464
NaN (Not-a-Number) value
causes, 147, 149
exceptions, 682
in floating point comparisons, 347
payload results, 333–334
Negation
logical, 327–328
numeric, 327
Nested array initializers, 631–632
Nested blocks, 104
Nested classes, 468
Nested members, 110–111

www.it-ebooks.info

Nested scopes, 120
Nested types, 498–499
  accessibility, 499–503
  description, 464
  fully qualified names for, 499
  in generic classes, 503
  member access contained by, 502–503
  partial, 482
  this access to, 500–501
Nesting
  aliases, 460
  with break, 431
  comments, 70
  hiding through, 124–125, 500
  object initializers, 308
New line escape sequence, 81
new modifier
  class members, 496
  classes, 468
  delegates, 672
  interface members, 640
  interfaces, 634
new operator
  anonymous objects, 317–319
  arrays, 53, 55, 312–315
  collection initializers, 310–312
  constructors, 43
  delegates, 315–317
  hidden methods, 126
  object initializers, 307–310
  objects, 305
  structs, 52
No fall through rule, 416-417
No side effects convention, 552
Non-nested types, 498
Non-nullable value type, 152
Non-virtual methods, 35
Nonterminal symbols, 65–66
Normal form function members, 272
Normalization Form C, 73
Not-a-Number (NaN) value
  causes, 147, 149
  exceptions, 682
  in floating point comparisons, 347
  payload results, 333–334
Notation, grammar, 65-67
NotSupportedException class, 593
Null coalescing operator, 360-361
Null field for events, 48
Null literals, 84, 152, 199
Null pointers, 714
Null-termination of strings, 733
Null values
  for array elements, 54
  in classes *vs.* structs, 613
  escape sequence for, 81
  garbage collector for, 134
Nullable boolean logical operators, 357
Nullable types, 11–12
  contents, 13
  conversions
    explicit, 207–208
    implicit, 198–199
    operators, 353–355
  description, 8
  equality operators with, 352
  overview, 151–152
NullReferenceException class
  array access, 300
  with as operator, 355
  delegate creation, 316
  delegate invocation, 678
  description, 685
  foreach statement, 427
  throw statement, 436
  unboxing conversions, 159
Numeric conversions
  explicit, 205–207
  implicit, 197
Numeric promotions, 244–246

O

object class, 141, 154
Object variables, 12–13
Objects
  creation expressions for
    definite assignment rules, 187
    new operator, 305-307
  deallocating, 22
  description, 139
  initializers, 307–310
  as instance types, 153

www.it-ebooks.info

Obsolete attribute, 705–706
Octal literals, 77
OnChanged method, 42, 48
One-dimensional arrays, 54
Open types, 162
Operands, 13, 238
Operation class, 35–36
Operator notation, 241
Operators, 13, 42, 49, 84-85
  arithmetic. *See* Arithmetic operators
  assignment operators, 16, 389
    compound, 393–394
    event, 394–395
    simple, 390–393
  binary. *See* Binary operators
  conditional, 361–363
  conversion, 575–578, 759
  declaration, 571–573
  enums, 668–669
  in ID string format, 759
  invocation, 254
  lexical grammar, 773
  lifted, 246–247
  logical, 355–357
  null coalescing, 360–361
  numeric promotions, 244–246
  operator !=, 49
  operator ==, 49–50
  overloading, 240–243
  overview, 238
  precedence and associativity, 238-240
  relational. *See* Relational operators
  shift, 343–344
  type-testing, 352–353
  unary. *See* Unary operators
Optional parameters, 522, 585–586
Optional symbols in grammar notation, 66
OR operators, 15
Order
  declaration, 103
  execution, 137–138
orderby clauses, 375, 380–384
Out property, 553
Outer variables, 369–373
OutOfMemoryException class, 313, 316, 339, 685
Output parameters, 30, 173, 526-527
Output-safe types, 636
Output types in type inference, 263
Output-unsafe types, 636
Overflow checking context, 322–325, 443
OverflowException class
  addition, 338
  arrays, 313
  checked operator, 323–324
  decimal type, 150
  description, 686
  division, 335
  increment and decrement operators, 329
  multiplication, 332–333
  remainder operator, 336
Overload resolution
  anonymous functions, 368
  function members, 270–275
Overloaded operators, 13
  purpose, 238
  shift, 343
Overloading
  indexers, 47
  methods, 38–40
  operators, 240–243
  signatures in, 38, 117-120
Overridden base methods, 535
Override events, 566
Override indexers, 567
Override methods, 535–537
Overriding
  event declarations, 566
  methods, 35
  property accessors, 46, 557
  property declarations, 557–558

P

Padding for pointers, 727
Paint method, 56, 539
Pair class, 24
Pair-wise declarations, 575
<para> tag, 748
Paragraph-separator characters, 69

www.it-ebooks.info

<param> tag, 742, 748
Parameter lists, variant type, 635–637
Parameters
  anonymous functions, 365
  arrays, 528–531
  attributes, 690–691
  entry points, 99
  function member invocations, 255–256
  indexers, 46–47, 567–568
  instance constructors, 581, 585–586
  methods, 29–32
    declaration, 522–525
    types, 524–531
  optional, 585–586
  output, 173, 526–527
  in overloading, 117–118
  reference, 172, 525–526
  type. *See* Type parameters
  value, 171, 525
<paramref> tag, 749
params modifier, 31–32, 528–531
Parentheses ()
  anonymous functions, 365
  in grammar notation, 66
  in ID string format, 755
  for operator precedence, 240
Parenthesized expressions, 283
Partial methods, 541
partial modifier, 471
  interfaces, 634
  structs, 609
  types, 481–482
Partial types, 471
  attributes, 482–483
  base classes, 484
  base interfaces, 484
  members, 485
  methods, 486–490
  modifiers, 483
  name binding, 490
  overview, 481–482
  type parameters and constraints, 483–484
Patterns, query expression, 387–389
Percent signs (%)
  assignment operators, 389
  remainder operator, 336–337
Periods (.)
  base access, 302
  members, 105
<permission> tag, 749
Permitted user-defined conversions, 214–215
Phases, type inference, 262
Plus (+) operator, 326
Plus signs (+)
  addition, 337–340
  assignment operators, 389
  increment operator, 303–305, 328–330
  pointers, 725
Point class
  base class, 25
  coordinates, 308
  declaration, 22
  instantiated objects, 51
  properties, 554
  source code, 760–762
Point struct, 611–612
  assignment operators, 391–392
  default values, 613
  field initializers, 616–618
  instantiated objects, 51–52
Point3D class, 25
Pointers
  arithmetic, 725–726
  arrays, 719–720
  conversions, 717–720
  element access, 723
  in expressions, 720–727
  for fixed variables, 728–733
  function, 671
  indirection, 716, 721
  member access, 721–722
  operators
    address-of, 724–725
    comparison, 726
    increment and decrement, 725
    sizeof, 727
  types, 713–716
  unsafe, 7, 709
  variables with, 716–717
Polymorphism, 22, 26
Pop method, 4–5
Positional parameters, 690–691

www.it-ebooks.info

Postfix increment and decrement operators, 303–305
#pragma directive, 96
#pragma warning directive, 96–97
Precedence of operators, 13, 238–240
Prefix increment and decrement operators, 328–330
Preprocessing directives
  conditional compilation, 87, 90–93
  declaration, 88–89
  diagnostic, 93–94
  lexical grammar, 774–777
  line, 95–96
  overview, 85–87
  pragma, 96–97
  preprocessing expressions, 87–88
  region, 94
Preprocessing expressions, 87–88
Primary expressions
  anonymous method, 326
  checked and unchecked operators, 322–325
  default value, 325
  element access, 298–301
  forms of, 278–279
  invocation, 288–298
  literals in, 279
  member access, 283–288
  new operator in
    anonymous objects, 317–319
    arrays, 312–315
    collection initializers, 310–312
    delegates, 315–317
    object initializers, 307–310
    objects, 305–307
  parenthesized, 283
  postfix increment and decrement operators, 303–305
  simple names in, 279–283
  this access in, 301–302
  typeof operator, 319–322
Primary operators, 14
PrintColor method, 58
Private accessibility, 23–24,107
Private constructors, 584–585
Productions, grammar,65
Program class, 47, 602–603, 614–615
Program structure, 4–6
Programs, 4, 65
Projection initializers, 319
Promotions, numeric, 244–246
Propagation, exception, 437
Properties, 4
  access to, 252
  accessibility, 555–556
  automatically implemented, 553–555
  declarations, 545–546
  example, 42
  in ID string format, 754,758
  indexers, 568
  interface, 641–642
  member names reserved for, 504–505
  overview, 43–46
  static and instance, 546
Property accessors, 46
  declarations, 547
  overview, 547–553
  types of, 553
Protected accessibility, 23–24
  declared, 107
  instance members, 113–116
  internal, 23–24, 107
Public accessibility, 23–24,107
Punctuators
  lexical grammar,773
  list of, 84–85
PurchaseTransaction class,92
Push method, 4

Q

Qualifiers, alias, 464–466
Query expressions
  ambiguities in, 376
  overview, 373–375
  patterns, 387–389
  translations in, 376–387
Question marks (?)
  null coalescing operator, 360–361
  ternary operators, 191, 361–362
Quotes (',") for characters, 80–81

www.it-ebooks.info

R

\r escape sequence, 81
Range variables, 375, 379
Rank of arrays, 54, 625-626
Reachability
  blocks, 401
  do statements, 421
  for statements, 424
  labeled statements, 406–407
  overview, 400–402
  return statements, 435
  statement lists, 403
  throw statements, 437
  while statements, 420–421
Read-only fields, 27–28, 511-513
Read-only properties, 45, 549–550, 554
Read-write properties, 45, 549–550
readonly modifier, 27, 511
ReadOnlyPoint class, 554
Reads, volatile, 514
Real literals, 78–79
ReAlloc method, 739
Recommended tags for comments, 743–753
Rectangle class, 308–309
Rectangle struct, 392
ref modifier, 30
Reference conversions
  explicit, 208–210
  implicit, 199–201
Reference parameters, 29–30, 172, 525–526
Reference types, 6–8, 152–153
  array, 53, 155
  class, 153–154
  constraints, 476
  delegate, 155
  dynamic, 154
  equality operators, 349–351
  interface, 155
  object, 154
  string, 154
References, 139
  parameter passing by, 29-30
  variable, 192–193
Referencing static class types, 470-471
Referent types, pointer, 713
Region directives, 94
Regular string literals, 81–82
Reimplementation, interface, 659–660
Relational operators
  booleans, 348
  decimal numbers, 348
  delegates, 351–352
  descriptions, 15
  enumerations, 348
  integers, 346
  lifted, 247
  overview, 344–345
  reference types, 349–351
  strings, 351
Release semantics, 514
Remainder operator, 336–337
<remarks> tag, 750
remove accessors
  attributes, 695
  events, 49, 564
RemoveEventHandler method, 565
Removing delegates, 342
Required parameters, 522
Reserved attributes, 699–700
  AttributeUsage, 700
  Conditional, 701–705
  Obsolete, 705–706
Reserved names for class members, 504-506
Reset method, 604
Resolution
  function members, 270–275
  operator overload, 38, 243
Resources, using statement for, 445-449
return statement
  definite assignment rules, 182–183
  example, 19
  methods, 33
  overview, 435
  with yield, 449–452
Return type
  entry points, 100
  inferred, 267–269
  methods, 28, 521–522
<returns> tag, 750
Right-associative operators, 239
Right shift operator, 343-344

www.it-ebooks.info

Rounding, 150
Rules for definite assignment, 178–192
running state for enumerator objects, 593–596
Runtime processes
  argument list evaluation, 257–259
  array creation, 313
  attribute instance retrieval, 699
  binding, 235
  delegate creation, 316
  function member invocations, 276–277
  increment and decrement operators, 304
  object creation, 306–307
  prefix increment and decrement operations, 329
  unboxing conversions, 160
Runtime types, 35, 532
RuntimeWrappedException class, 439

S

sbyte type, 9
Scopes
  aliases, 459–460
  attributes, 694
  vs. declaration space, 101
  local variables, 410
  for name hiding, 124–127
  overview, 120–124
Sealed accessors, 557
Sealed classes, 469, 474–475
Sealed events, 566
Sealed indexers, 567
Sealed methods, 537–538
sealed modifier, 469, 537–538
Sections for attributes, 692
<see> tag, 751
<seealso> tag, 751–752
select clauses, 375, 378, 384–385
Selection statements, 413
  if, 413–414
  switch, 414–419
Semicolons (;)
  accessors, 548
  interface identifiers, 642
  method bodies, 544
  namespace declarations, 454
Sequences in query expressions, 375
set accessors
  for attributes, 695
  defined, 45
  description, 557
  working with, 547–550
SetItems method, 56
SetNextSerialNo method, 34
SetText method, 56
Shape class, 539
Shift operators
  described, 15
  overview, 343–344
Short-circuiting logical operators, 358
short type, 9–10, 144
ShowHelp method, 62
Side effects
  with accessors, 552
  and execution order, 137–138
Signatures
  anonymous functions, 365–366
  indexers, 568
  methods, 28, 521
  operators
    binary, 575
    conversion, 578
    unary, 573
  in overloading, 38, 117–120
Signed integrals, 8–9
Simple assignment
  definite assignment rules, 188
  operator, 389
  overview, 390–393
Simple expression assignment rules, 186
Simple names
  in primary expressions, 279–283
  and type names, 286–287
Simple types, 8, 140–144
Single-dimensional arrays
  defined, 625
  example, 54
  initializers, 631
Single-line comments, 69–70, 741–742
Single quotes (') for characters, 80–81
Single-use attribute classes, 688

www.it-ebooks.info

SizeOf method, 739
sizeof operator, 727
Slashes (/)
  assignment operators, 389
  comments, 69–70, 741–742
  division, 334–335
Slice method, 542–543
Source files
  compilation, 6
  described, 65
  Point class, 760–762
Source types in conversions, 215
SplitPath method, 527
SqlBoolean struct, 621
SqlInt32 struct, 621
Square brackets ([])
  arrays, 11, 54
  attributes, 692
  indexers, 46
  pointers, 716, 723
Square method, 60
Squares class, 33
Stack
  allocation, 736–738
  values on, 7
Stack class, 4–5, 597–598
stackalloc operator, 716, 736–738
StackOverflowException class, 686, 737
Standard conversions, 213–214
Startup, application, 99–100
Statement lists, 403–404
Statements, 399–400
  blocks in, 402–404
  checked and unchecked, 443
  declaration, 407–412
  definite assignment rules, 179
  empty, 404–406
  end points and reachability, 400-402
  expression, 17, 179, 412–413
  in grammar notation, 66
  iteration, 420
    do, 421
    for, 422–423
    foreach, 423–429
    while, 420–421
  jump, 429–431
    break, 431
    continue, 432
    goto, 433–434
    return, 435
    throw, 436–437
  labeled, 406–407
  lock, 443–445
  overview, 16–21
  selection, 413
    if, 413–414
    switch, 414–419
  syntactic grammar, 788–793
  try, 438–443
  using, 445–449
  yield, 449–452
States, definite assignment, 178
Static binding, 234–235
Static classes, 470–471
Static constructors, 42
  in classes *vs.* structs, 619
  overview, 586–589
Static events, 565
Static fields, 26, 510–511
  for constants, 512–513
  initialization, 515–519
  read-only, 511-513
Static members, 22, 496–498
Static methods, 28
  garbage collection, 133
  vs. instance, 33–34, 531
static modifier, 470–471
Static properties, 546
Static variables, 170, 510–511
Status codes, termination, 100
String class, 39–40, 154
string type, 9, 154
StringFromColor method, 667
StringListEvent method, 639
Strings
  concatenation, 339
  equality operators, 351
  literals, 81–84
  null-termination, 733
  switch governing type, 418

www.it-ebooks.info

Structs
assignment, 612
boxing and unboxing, 613–616
vs. classes, 610–619
constructors, 617–618
declarations, 608–609
default values, 612–613
destructors, 619
examples
database boolean type, 622–623
database integer type, 619–621
field initializers in, 616–617
inheritance, 612
instance variables, 171
interface implementation by, 57
members, 105–106, 609
overview, 50–53, 607
syntactic grammar, 803–804
this access in, 616
types, 6, 8, 10–11, 143
value semantics, 610–612
Subtraction operator, 340–342
Suffixes, numeric, 76–79
<summary> tag, 742, 752
SuppressFinalize method, 101
suspended state, 593–596
Swap method, 29
switch statement
definite assignment rules, 180
example, 18
overview, 414–419
reachability, 402
Syntactic grammar, 67
arrays, 804–805
attributes, 807–809
basic concepts, 777
classes, 794–803
delegates, 807
enums, 806–807
expressions, 779–788
interfaces, 805–806
namespaces, 793–794
statements, 788–793
structs, 803–804
types, 777–779
variables, 779
System-level exceptions, 682
System namespace, 143

T

\t escape sequence, 81
Tab escape sequence, 81
Tags for comments, 743–753
Target types in conversions, 215
Targets
goto, 433–434
jump, 430
Terminal symbols, 65-66
Termination, application, 100–101
Terminators, line, 68–69, 767
Ternary operators, 238, 361-363
TextReader class, 449
TextWriter class, 449
this access
classes *vs.* structs, 616
indexers, 46
instance constructors, 585–586
nested types, 500–501
overview, 301–302
properties, 546
static methods, 33
Thread-safe delegates, 677
Three-dimensional arrays, 54
Throw points, 437
throw statement
definite assignment rules, 182
example, 20
for exceptions, 683
overview, 436–437
Tildes (~)
bitwise complement, 328
conversion, 759
Time, binding, 235
ToInt32 method, 542
Tokens, 71
identifiers, 72–74
keywords, 74–75

www.it-ebooks.info

lexical grammar,769
literals, 76–84
operators, 84–85
unicode character escape sequence, 71–72
ToString method,339
and boxing, 614–615
DBBool, 623
DBInt, 621
Point, 761–762
Translate method,761
Translations in query expressions, 376-387
Transparent identifiers in query expressions, 377, 385–387
Tree class, 602-603
Tree types, expression, 165-166
Trig class,585
True value,76
try statement
definite assignment rules, 183–185
example, 20
for exceptions, 684–685
with goto, 434
overview, 438–443
TryParse method,527
Two-dimensional arrays
allocating, 54
initializers, 631
Type casts,59
Type inference, 259-270
Type names, 127-130
fully qualified, 131
identical, 286–287
Type parameters,139
class declarations, 24–25, 471–472
constraints, 475–481
conversions, 211–212
implicit conversions, 203–204
overview, 164–165
partial types, 483–484
Type-safe design,1
Type testing operators
as, 353–355
described, 15
is, 352–353
TypeInitializationException class, 684,686
typeof operator
pointers with,713
primary expressions, 319–322
<typeparam> tag,753
<typeparamref> tag,753
Types
aliases for, 456-461
attribute parameter,691
boxing and unboxing, 156–158
constructed, 160–164, 493–494
declarations, 10,464
dynamic, 166–167
in ID string format, 754-756
importing, 461–463
instance, 492
nested, 464, 498–504
nullable. *See* Nullable types
overview, 6–13,139
partial. *See* Partial types
pointer. *See* Pointers
reference. *See* Reference types
syntactic grammar, 777–779
underlying, 58–59, 151
value. *See* Value types

U

uint type, 10
ulong type, 10
Unary operators, 326
cast expressions, 330–331
described, 14, 238
in ID string format,759
lifted, 246
minus, 327
numeric promotions, 244
overload resolution, 242–243
overloadable, 240–241
overview, 573–574
plus, 326
prefix increment and decrement, 328-330
Unassigned variables, 177
Unbound types, 160,162

www.it-ebooks.info

Unboxing conversions
described, 210
overview, 158–160
Unboxing operations
in classes *vs.* structs, 613–616
example, 12
unchecked statement
definite assignment rules, 179
example, 20
overview, 443
in primary expressions, 322–325
#undef directive, 87–89
Undefined conditional compilation symbols, 87
Underlying types
enums, 58–59, 664
nullable, 151
Underscore characters (_) for identifiers, 72–74
Unicode characters
escape sequence, 71–72
lexical grammar, 67, 769
for strings, 9
Unicode Normalization Form C, 73
Unified type system, 1
Uniqueness
aliases, 466
interface implementations, 650–652
Unmanaged types, 713
Unreachable statements, 400
Unsafe code, 709
contexts in, 710–713
dynamic memory allocation, 738–740
fixed-size buffers, 733–736
fixed statement, 728–733
grammar extensions for, 809-812
pointers
arrays, 719–720
conversions, 717–720
in expressions, 720–727
support for, 7
types, 713–716
stack allocation, 736–738
unsafe modifier, 710–713
Unsigned integrals, 8–10
Unwrapping non-nullable value types, 152
Upper-bound type inferences, 265–266
User-defined conversions, 214
evaluation, 215–216
explicit, 213, 218–219
implicit, 204, 217
lifted operators, 215
overview, 575–578
permitted, 214–215
User-defined operators
candidate, 243
conditional logical, 359–360
ushort type, 10
Using directives
for aliases, 458–461
definite assignment rules, 185–186
example, 21
for importing types, 461–463
overview, 445–449, 457
purpose, 3

V

\v escape sequence, 81
Value method, 620
Value parameters, 29, 171, 525
Value property, 152
<value> tag, 752
Value types
bool, 150–151
constraints, 476
contents, 13
decimal, 149–150
default constructors, 141–142
described, 8
enumeration, 151
floating point, 146–149
integral, 145–146
nullable, 151–152
overview, 140–141
simple, 143–144
storing, 7
struct, 143

www.it-ebooks.info

Values
array types, 626
classes *vs.* structs, 610–612
default, 141
classes *vs.* structs, 612–613
initialization, 175–176
enums, 668–669
expressions, 233
fields, 510–511
local constants, 412
variables, 169, 175–176, 408–409
ValueType class, 141,612
VariableReference class, 35–36
Variables, 169
anonymous functions, 369–373
array elements, 171
declarations, 175, 407–411
default values, 175–176
definite assignment. *See* Definite assignment
fixed addresses for, 728–733
fixed and moveable, 716–717
initializers, 516–519, 581
instance, 170–171, 510–511
local, 173–175
in methods, 32–33
names, 170
output parameters, 173
overview, 12–13
query expressions, 375, 379
reference parameters, 172
references, 192–193
scope, 124–125, 410
static, 170, 510–511
syntactic grammar, 779
value parameters, 171
Variant type parameter lists, 635–637
Verbatim identifiers, 74
Verbatim string literals, 81–83
Versioning
of constants, 512–513
described, 1
Vertical bars (|)
assignment operators, 389
definite assignment rules, 189–190
logical operators, 355–359
preprocessing expressions, 87
Vertical tab escape sequence, 81
Vexing exceptions, 686
Viewers, documentation, 741
Virtual accessors, 46, 557–559
Virtual events, 566
Virtual indexers, 567
Virtual methods
description, 236
overview, 35–38
working with, 532–534
Visibility in scope, 120, 124
void type and values
entry point method, 100
events, 564
pointers, 714
return, 28, 33
with typeof, 320
Volatile fields, 514–515

W

WaitForPendingFinalizers method, 136
#warning directive, 94
warnings, preprocessing directives, 96–97
where clauses
query expressions, 380–384
type parameter constraints, 163, 476
while statement
definite assignment rules, 181
example, 18
overview, 420–421
Whitespace
in comments, 742
defined, 70–71
in ID string format, 754
lexical grammar, 769
Win32 component interoperability, 707
workCompleted method, 47
Worker class, 47–48
Wrapping non-nullable value types, 152
Write method, 31

www.it-ebooks.info

Write-only properties, 45, 549–550,554
WriteLine method, 3, 31,136
Writes, volatile,514

X

\x escape sequence,80
XAttribute class, 696–697
XML (Extensible Markup Language), 741–742, 762–765
XOR operators,15

Y

yield statement
  definite assignment rules,186
  example,20
  overview, 449–452
  yield break, 594–595
  yield return, 594–595
Yield type iterators,592

Z

Zero values, 146–148

www.it-ebooks.info