Special Annotated Edition for C# 4.0

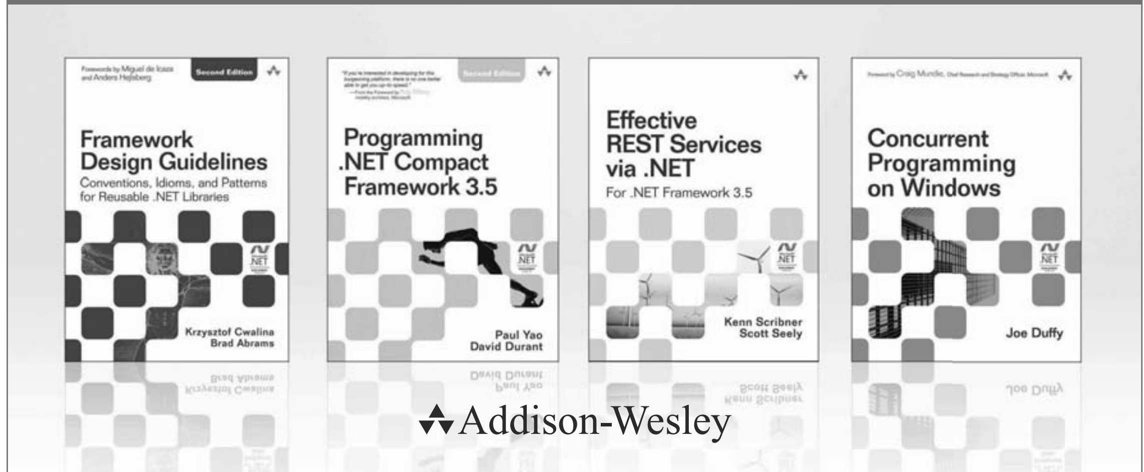
The C# Programming Language

Fourth Edition

Anders Hejlsberg
Mads Torgersen
Scott Wiltamuth
Peter Golde

The C# Programming Language
Fourth Edition

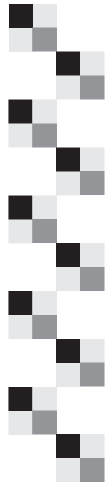
Microsoft® .NET Development Series



Visit **informit.com/msdotnetseries** for a complete list of available products.

The award-winning **Microsoft .NET Development Series** was established in 2002 to provide professional developers with the most comprehensive, practical coverage of the latest .NET technologies. Authors in this series include Microsoft architects, MVPs, and other experts and leaders in the field of Microsoft development technologies. Each book provides developers with the vital information and critical insight they need to write highly effective applications.

Addison-Wesley | Cisco Press | EXAM/CRAM | IBM Press | QUE | PRENTICE HALL | SAMS | Safari Books Online



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

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

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

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

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

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

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

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

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

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

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.

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

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.

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.

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.

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.

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.

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#.

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;
```

```
            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);
```

```
            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

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.

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.

> **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

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`.

> **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.

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.

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.

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`

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

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.

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.

<table>
<tr><th>Statement</th><th>Example</th></tr>
<tr><td>Local variable declaration</td><td><pre>static void Main() {
    int a;
    int b = 2, c = 3;
    a = 1;
    Console.WriteLine(a + b + c);
}</pre></td></tr>
<tr><td>Local constant declaration</td><td><pre>static void Main() {
    const float pi = 3.1415927f;
    const int r = 25;
    Console.WriteLine(pi * r * r);
}</pre></td></tr>
<tr><td>Expression statement</td><td><pre>static void Main() {
    int i;
    i = 123;   // Expression statement
    Console.WriteLine(i);   // Expression statement
    i++;   // Expression statement
    Console.WriteLine(i);   // Expression statement
}</pre></td></tr>
</table>

Continued

1. Introduction

<table>
<tr><th>Statement</th><th>Example</th></tr>
<tr><td><code>if</code> statement</td><td><pre><code>static void Main(string[] args) {
    if (args.Length == 0) {
        Console.WriteLine("No arguments");
    }
    else {
        Console.WriteLine("One or more arguments");
    }
}</code></pre></td></tr>
<tr><td><code>switch</code> statement</td><td><pre><code>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;
        }
    }
}</code></pre></td></tr>
<tr><td><code>while</code> statement</td><td><pre><code>static void Main(string[] args) {
    int i = 0;
    while (i < args.Length) {
        Console.WriteLine(args[i]);
        i++;
    }
}</code></pre></td></tr>
<tr><td>do statement</td><td><pre><code>static void Main() {
    string s;
    do {
        s = Console.ReadLine();
        if (s != null) Console.WriteLine(s);
    } while (s != null);
}</code></pre></td></tr>
</table>

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

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
    }
}
```

<table>
<tr><th>Statement</th><th>Example</th></tr>
<tr><td><code>lock</code> statement</td><td><pre>class Account
{
    decimal balance;
    public void Withdraw(decimal amount) {
        lock (this) {
            if (amount > balance) {
                throw new Exception("Insufficient funds");
            }
            balance -= amount;
        }
    }
}</pre></td></tr>
<tr><td><code>using</code> statement</td><td><pre>static void Main() {
    using (TextWriter w = File.CreateText("test.txt")) {
        w.WriteLine("Line one");
        w.WriteLine("Line two");
        w.WriteLine("Line three");
    }
}</pre></td></tr>
</table>

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 }.

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.

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!

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.

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`

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:

```
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`.

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.

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"
    }
}
```

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.

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);
```

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.

```
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()
    {
```

```
            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.

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 ***non-virtual 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);
}
```

```
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");
    }
}
```

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"
    }
}
```

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)");
    }
```

```
    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

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.

```
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

```
    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.

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.

■ **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 {
```

```
        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.

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();
}
```

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

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++;
    }
```

```
        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);
```

```
            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`.

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.

■ **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.

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.

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.

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.

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.

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.

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,
```

```
        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!

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[]`.

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; }
    }
```

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

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#.

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.

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

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

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.