



DRAFT INTERNATIONAL STANDARD ISO/IEC 23270

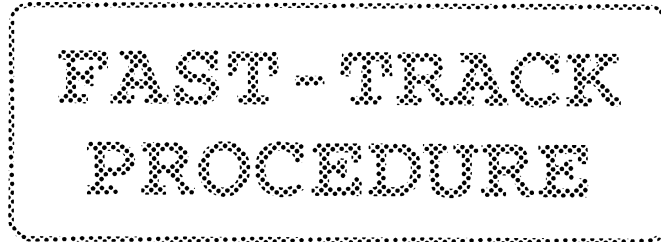
Attributed to ISO/IEC JTC 1 by the Central Secretariat (see page ii)

Voting begins on
2002-02-14

Voting terminates on
2002-07-14

L2/02-119

INTERNATIONAL ORGANIZATION FOR STANDARDIZATION • МЕЖДУНАРОДНАЯ ОРГАНИЗАЦИЯ ПО СТАНДАРТИЗАЦИИ • ORGANISATION INTERNATIONALE DE NORMALISATION
INTERNATIONAL ELECTROTECHNICAL COMMISSION • МЕЖДУНАРОДНАЯ ЭЛЕКТРОТЕХНИЧЕСКАЯ КОММИСИЯ • COMMISSION ÉLECTROTECHNIQUE INTERNATIONALE



Information technology — C# Language Specification

Technologies de l'information — Spécification du langage C#

ICS 35.060

In accordance with the provisions of Council Resolution 21/1986 this DIS is circulated in the English language only.

Conformément aux dispositions de la Résolution du Conseil 21/1986, ce DIS est distribué en version anglaise seulement.

THIS DOCUMENT IS A DRAFT CIRCULATED FOR COMMENT AND APPROVAL. IT IS THEREFORE SUBJECT TO CHANGE AND MAY NOT BE REFERRED TO AS AN INTERNATIONAL STANDARD UNTIL PUBLISHED AS SUCH.

IN ADDITION TO THEIR EVALUATION AS BEING ACCEPTABLE FOR INDUSTRIAL, TECHNOLOGICAL, COMMERCIAL AND USER PURPOSES, DRAFT INTERNATIONAL STANDARDS MAY ON OCCASION HAVE TO BE CONSIDERED IN THE LIGHT OF THEIR POTENTIAL TO BECOME STANDARDS TO WHICH REFERENCE MAY BE MADE IN NATIONAL REGULATIONS.



NOTE FROM ITTF

This draft International Standard is submitted for JTC 1 national body vote under the Fast-Track Procedure.

In accordance with Resolution 30 of the JTC 1 Berlin Plenary 1993, the proposer of this document recommends assignment of ISO/IEC 23270 to JTC 1/SC 22.

"FAST-TRACK" PROCEDURE

1 Any P-member and any Category A liaison organization of ISO/IEC JTC 1 may propose that an existing standard from any source be submitted directly for vote as a DIS. The criteria for proposing an existing standard for the fast-track procedure are a matter for each proposer to decide.

2 The proposal shall be received by the ITTF which will take the following actions.

2.1 To settle the copyright and/or trade mark situation with the proposer, so that the proposed text can be freely copied and distributed within JTC 1 without restriction.

2.2 To assess in consultation with the JTC 1 secretariat which SC is competent for the subject covered by the proposed standard and to ascertain that there is no evident contradiction with other International Standards.

2.3 To distribute the text of the proposed standard as a DIS. In case of particularly bulky documents the ITTF may demand the necessary number of copies from the proposer.

3 The period for combined DIS voting shall be six months. In order to be accepted the DIS must be supported by 75 % of the votes cast (abstention is not counted as a vote) and by two-thirds of the P-members voting of JTC 1.

4 At the end of the voting period, the comments received, whether editorial only or technical, will be dealt with by a working group appointed by the secretariat of the relevant SC.

5 If, after the deliberations of this WG, the requirements of 3 above are met, the amended text shall be sent to the ITTF by the secretariat of the relevant SC for publication as an International Standard.

If it is impossible to agree to a text meeting the above requirements, the proposal has failed and the procedure is terminated.

In either case the WG shall prepare a full report which will be circulated by the ITTF.

6 If the proposed standard is accepted and published, its maintenance will be handled by JTC 1.

ECMA

Standardizing Information and Communication Systems

C# Language Specification

Brief history

This ECMA Standard is based on a submission from Hewlett-Packard, Intel, and Microsoft, that describes a language called C#, which was developed within Microsoft. The principal inventors of this language were Anders Hejlsberg, Scott Wiltamuth, and Peter Golde. The first widely distributed implementation of C# was released by Microsoft in July 2000, as part of its .NET Framework initiative.

ECMA Technical Committee 39 (TC39) Task Group 2 (TG2) was formed in September 2000, to produce a standard for C#. Another Task Group, TG3, was also formed at that time to produce a standard for a library and execution environment called Common Language Infrastructure (CLI). (CLI is based on a subset of the .NET Framework.) Although Microsoft's implementation of C# relies on CLI for library and runtime support, other implementations of C# need not, provided they support an alternate way of getting at the minimum CLI features required by this C# standard.

As the definition of C# evolved, the goals used in its design were as follows:

- C# is intended to be a simple, modern, general-purpose, object-oriented programming language.
- The language, and implementations thereof, should provide support for software engineering principles such as strong type checking, array bounds checking, detection of attempts to use uninitialized variables, and automatic garbage collection. Software robustness, durability, and programmer productivity are important.
- The language is intended for use in developing software components suitable for deployment in distributed environments.
- Source code portability is very important, as is programmer portability, especially for those programmers already familiar with C and C++.
- Support for internationalization is very important.
- C# is intended to be suitable for writing applications for both hosted and embedded systems, ranging from the very large that use sophisticated operating systems, down to the very small having dedicated functions.
- Although C# applications are intended to be economical with regards to memory and processing power requirements, the language was not intended to compete directly on performance and size with C or assembly language.

The development of this standard started in November 2000.

It is intended that the final version of this ECMA Standard will be submitted to ISO/IEC JTC 1 for adoption under its fast-track procedure.

It is expected there will be future revisions to this Standard, primarily to add new functionality.

Adopted as an ECMA Standard by the General Assembly of December 2001.

1		
2		
	Table of Contents	
3	1. Scope	1
4	2. Conformance	3
5	3. References	5
6	4. Definitions	7
7	5. Notational conventions	9
8	6. Acronyms and abbreviations	11
9	7. General description	13
10	8. Language Overview	15
11	8.1 Getting started.....	15
12	8.2 Types.....	16
13	8.2.1 Predefined types	17
14	8.2.2 Conversions.....	19
15	8.2.3 Array types	19
16	8.2.4 Type system unification	21
17	8.3 Variables and parameters	22
18	8.4 Automatic memory management.....	25
19	8.5 Expressions	27
20	8.6 Statements.....	27
21	8.7 Classes	30
22	8.7.1 Constants	31
23	8.7.2 Fields.....	32
24	8.7.3 Methods.....	33
25	8.7.4 Properties.....	34
26	8.7.5 Events.....	35
27	8.7.6 Operators	36
28	8.7.7 Indexers	37
29	8.7.8 Instance constructors	38
30	8.7.9 Destructors	38
31	8.7.10 Static constructors	39
32	8.7.11 Inheritance.....	39
33	8.8 Structs	40
34	8.9 Interfaces.....	41
35	8.10 Delegates.....	42
36	8.11 Enums	43
37	8.12 Namespaces and assemblies	44
38	8.13 Versioning.....	45
39	8.14 Attributes	47
40	9. Lexical structure	49
41	9.1 Programs	49
42	9.2 Grammars.....	49
43	9.2.1 Lexical grammar	49
44	9.2.2 Syntactic grammar.....	49
45	9.3 Lexical analysis.....	50
46	9.3.1 Line terminators	50
47	9.3.2 Comments	50

1	9.3.3 White space	52
2	9.4 Tokens.....	52
3	9.4.1 Unicode escape sequences	52
4	9.4.2 Identifiers	53
5	9.4.3 Keywords	54
6	9.4.4 Literals.....	55
7	9.4.5 Operators and punctuators.....	60
8	9.5 Pre-processing directives	60
9	9.5.1 Conditional compilation symbols.....	61
10	9.5.2 Pre-processing expressions	61
11	9.5.3 Declaration directives.....	62
12	9.5.4 Conditional compilation directives	63
13	9.5.5 Diagnostic directives	65
14	9.5.6 Region control.....	65
15	9.5.7 Line directives	66
16	10. Basic concepts	67
17	10.1 Application startup.....	67
18	10.2 Application termination	67
19	10.3 Declarations	68
20	10.4 Members	70
21	10.4.1 Namespace members.....	70
22	10.4.2 Struct members.....	70
23	10.4.3 Enumeration members.....	71
24	10.4.4 Class members.....	71
25	10.4.5 Interface members	71
26	10.4.6 Array members	71
27	10.4.7 Delegate members	71
28	10.5 Member access.....	71
29	10.5.1 Declared accessibility.....	71
30	10.5.2 Accessibility domains	72
31	10.5.3 Protected access for instance members	74
32	10.5.4 Accessibility constraints.....	75
33	10.6 Signatures and overloading.....	76
34	10.7 Scopes	77
35	10.7.1 Name hiding.....	79
36	10.8 Namespace and type names	81
37	10.8.1 Fully qualified names	82
38	10.9 Automatic memory management.....	82
39	10.10 Execution order.....	85
40	11. Types.....	87
41	11.1 Value types	87
42	11.1.1 Default constructors	88
43	11.1.2 Struct types.....	88
44	11.1.3 Simple types	89
45	11.1.4 Integral types	89
46	11.1.5 Floating point types.....	90
47	11.1.6 The decimal type	92
48	11.1.7 The bool type.....	92
49	11.1.8 Enumeration types.....	92
50	11.2 Reference types.....	92
51	11.2.1 Class types.....	93
52	11.2.2 The object type.....	93
53	11.2.3 The string type.....	93

1	11.2.4 Interface types	94
2	11.2.5 Array types	94
3	11.2.6 Delegate types	94
4	11.3 Boxing and unboxing	94
5	11.3.1 Boxing conversions	94
6	11.3.2 Unboxing conversions	95
7	12. Variables.....	97
8	12.1 Variable categories.....	97
9	12.1.1 Static variables	97
10	12.1.2 Instance variables	97
11	12.1.3 Array elements	98
12	12.1.4 Value parameters	98
13	12.1.5 Reference parameters	98
14	12.1.6 Output parameters	98
15	12.1.7 Local variables	99
16	12.2 Default values	99
17	12.3 Definite assignment	100
18	12.3.1 Initially assigned variables	100
19	12.3.2 Initially unassigned variables	101
20	12.3.3 Precise rules for determining definite assignment	101
21	12.4 Variable references	109
22	12.5 Atomicity of variable references	109
23	13. Conversions	111
24	13.1 Implicit conversions.....	111
25	13.1.1 Identity conversion.....	111
26	13.1.2 Implicit numeric conversions	111
27	13.1.3 Implicit enumeration conversions	112
28	13.1.4 Implicit reference conversions	112
29	13.1.5 Boxing conversions	112
30	13.1.6 Implicit constant expression conversions.....	112
31	13.1.7 User-defined implicit conversions.....	113
32	13.2 Explicit conversions.....	113
33	13.2.1 Explicit numeric conversions	113
34	13.2.2 Explicit enumeration conversions	115
35	13.2.3 Explicit reference conversions	115
36	13.2.4 Unboxing conversions.....	115
37	13.2.5 User-defined explicit conversions	116
38	13.3 Standard conversions	116
39	13.3.1 Standard implicit conversions	116
40	13.3.2 Standard explicit conversions.....	116
41	13.4 User-defined conversions.....	116
42	13.4.1 Permitted user-defined conversions	116
43	13.4.2 Evaluation of user-defined conversions	116
44	13.4.3 User-defined implicit conversions.....	117
45	13.4.4 User-defined explicit conversions	118
46	14. Expressions.....	121
47	14.1 Expression classifications	121
48	14.1.1 Values of expressions.....	122
49	14.2 Operators.....	122
50	14.2.1 Operator precedence and associativity	122
51	14.2.2 Operator overloading.....	123
52	14.2.3 Unary operator overload resolution.....	124
53	14.2.4 Binary operator overload resolution.....	125

1	14.2.5 Candidate user-defined operators.....	125
2	14.2.6 Numeric promotions.....	125
3	14.3 Member lookup.....	127
4	14.3.1 Base types.....	127
5	14.4 Function members.....	127
6	14.4.1 Argument lists.....	130
7	14.4.2 Overload resolution.....	132
8	14.4.3 Function member invocation.....	134
9	14.5 Primary expressions.....	135
10	14.5.1 Literals.....	136
11	14.5.2 Simple names.....	136
12	14.5.3 Parenthesized expressions.....	137
13	14.5.4 Member access.....	137
14	14.5.5 Invocation expressions.....	139
15	14.5.6 Element access.....	141
16	14.5.7 This access.....	142
17	14.5.8 Base access.....	143
18	14.5.9 Postfix increment and decrement operators.....	143
19	14.5.10 The <code>new</code> operator.....	144
20	14.5.11 The <code>typeof</code> operator.....	148
21	14.5.12 The <code>checked</code> and <code>unchecked</code> operators.....	149
22	14.6 Unary expressions.....	151
23	14.6.1 Unary plus operator.....	152
24	14.6.2 Unary minus operator.....	152
25	14.6.3 Logical negation operator.....	152
26	14.6.4 Bitwise complement operator.....	153
27	14.6.5 Prefix increment and decrement operators.....	153
28	14.6.6 Cast expressions.....	154
29	14.7 Arithmetic operators.....	154
30	14.7.1 Multiplication operator.....	155
31	14.7.2 Division operator.....	155
32	14.7.3 Remainder operator.....	156
33	14.7.4 Addition operator.....	157
34	14.7.5 Subtraction operator.....	159
35	14.8 Shift operators.....	160
36	14.9 Relational and type-testing operators.....	161
37	14.9.1 Integer comparison operators.....	162
38	14.9.2 Floating-point comparison operators.....	163
39	14.9.3 Decimal comparison operators.....	163
40	14.9.4 Boolean equality operators.....	163
41	14.9.5 Enumeration comparison operators.....	164
42	14.9.6 Reference type equality operators.....	164
43	14.9.7 String equality operators.....	165
44	14.9.8 Delegate equality operators.....	165
45	14.9.9 The <code>is</code> operator.....	166
46	14.9.10 The <code>as</code> operator.....	166
47	14.10 Logical operators.....	167
48	14.10.1 Integer logical operators.....	167
49	14.10.2 Enumeration logical operators.....	167
50	14.10.3 Boolean logical operators.....	168
51	14.11 Conditional logical operators.....	168
52	14.11.1 Boolean conditional logical operators.....	168
53	14.11.2 User-defined conditional logical operators.....	169
54	14.12 Conditional operator.....	169
55	14.13 Assignment operators.....	170

1	14.13.1 Simple assignment.....	170
2	14.13.2 Compound assignment.....	172
3	14.13.3 Event assignment.....	173
4	14.14 Expression.....	173
5	14.15 Constant expressions.....	173
6	14.16 Boolean expressions.....	174
7	15. Statements	175
8	15.1 End points and reachability.....	175
9	15.2 Blocks	177
10	15.2.1 Statement lists	177
11	15.3 The empty statement	177
12	15.4 Labeled statements.....	178
13	15.5 Declaration statements.....	178
14	15.5.1 Local variable declarations.....	178
15	15.5.2 Local constant declarations	179
16	15.6 Expression statements.....	180
17	15.7 Selection statements.....	180
18	15.7.1 The <code>if</code> statement.....	180
19	15.7.2 The <code>switch</code> statement	181
20	15.8 Iteration statements	184
21	15.8.1 The <code>while</code> statement	184
22	15.8.2 The <code>do</code> statement.....	185
23	15.8.3 The <code>for</code> statement.....	185
24	15.8.4 The <code>foreach</code> statement.....	186
25	15.9 Jump statements.....	188
26	15.9.1 The <code>break</code> statement	189
27	15.9.2 The <code>continue</code> statement.....	190
28	15.9.3 The <code>goto</code> statement.....	190
29	15.9.4 The <code>return</code> statement	191
30	15.9.5 The <code>throw</code> statement	192
31	15.10 The <code>try</code> statement	193
32	15.11 The checked and unchecked statements.....	195
33	15.12 The <code>lock</code> statement	196
34	15.13 The <code>using</code> statement.....	196
35	16. Namespaces	199
36	16.1 Compilation units.....	199
37	16.2 Namespace declarations.....	199
38	16.3 Using directives	200
39	16.3.1 Using alias directives	201
40	16.3.2 Using namespace directives	203
41	16.4 Namespace members	204
42	16.5 Type declarations	205
43	17. Classes.....	207
44	17.1 Class declarations	207
45	17.1.1 Class modifiers.....	207
46	17.1.2 Class base specification.....	208
47	17.1.3 Class body	209
48	17.2 Class members	210
49	17.2.1 Inheritance.....	211
50	17.2.2 The <code>new</code> modifier	211
51	17.2.3 Access modifiers	212
52	17.2.4 Constituent types.....	212
53	17.2.5 Static and instance members	212

1	17.2.6 Nested types	213
2	17.2.7 Reserved member names	216
3	17.3 Constants	217
4	17.4 Fields	219
5	17.4.1 Static and instance fields	220
6	17.4.2 Readonly fields	220
7	17.4.3 Volatile fields	221
8	17.4.4 Field initialization	222
9	17.4.5 Variable initializers	223
10	17.5 Methods	225
11	17.5.1 Method parameters	226
12	17.5.2 Static and instance methods	231
13	17.5.3 Virtual methods	232
14	17.5.4 Override methods	234
15	17.5.5 Sealed methods	235
16	17.5.6 Abstract methods	236
17	17.5.7 External methods	237
18	17.5.8 Method body	237
19	17.5.9 Method overloading	238
20	17.6 Properties	238
21	17.6.1 Static and instance properties	239
22	17.6.2 Accessors	240
23	17.6.3 Virtual, sealed, override, and abstract accessors	244
24	17.7 Events	245
25	17.7.1 Field-like events	247
26	17.7.2 Event accessors	248
27	17.7.3 Static and instance events	249
28	17.7.4 Virtual, sealed, override, and abstract accessors	249
29	17.8 Indexers	250
30	17.8.1 Indexer overloading	253
31	17.9 Operators	253
32	17.9.1 Unary operators	254
33	17.9.2 Binary operators	255
34	17.9.3 Conversion operators	255
35	17.10 Instance constructors	257
36	17.10.1 Constructor initializers	258
37	17.10.2 Instance variable initializers	258
38	17.10.3 Constructor execution	259
39	17.10.4 Default constructors	260
40	17.10.5 Private constructors	261
41	17.10.6 Optional instance constructor parameters	261
42	17.11 Static constructors	261
43	17.12 Destructors	263
44	18. Structs	265
45	18.1 Struct declarations	265
46	18.1.1 Struct modifiers	265
47	18.1.2 Struct interfaces	265
48	18.1.3 Struct body	266
49	18.2 Struct members	266
50	18.3 Class and struct differences	266
51	18.3.1 Value semantics	266
52	18.3.2 Inheritance	267
53	18.3.3 Assignment	267
54	18.3.4 Default values	267

1	18.3.5 Boxing and unboxing	268
2	18.3.6 Meaning of <code>this</code>	268
3	18.3.7 Field initializers.....	268
4	18.3.8 Constructors	268
5	18.3.9 Destructors	269
6	18.4 Struct examples.....	269
7	18.4.1 Database integer type	269
8	18.4.2 Database boolean type.....	271
9	19. Arrays	273
10	19.1 Array types.....	273
11	19.1.1 The <code>System.Array</code> type.....	274
12	19.2 Array creation	274
13	19.3 Array element access	274
14	19.4 Array members	274
15	19.5 Array covariance.....	274
16	19.6 Array initializers	275
17	20. Interfaces	277
18	20.1 Interface declarations	277
19	20.1.1 Interface modifiers	277
20	20.1.2 Base interfaces.....	277
21	20.1.3 Interface body.....	278
22	20.2 Interface members.....	278
23	20.2.1 Interface methods	279
24	20.2.2 Interface properties.....	279
25	20.2.3 Interface events	280
26	20.2.4 Interface indexers	280
27	20.2.5 Interface member access	280
28	20.3 Fully qualified interface member names.....	282
29	20.4 Interface implementations.....	282
30	20.4.1 Explicit interface member implementations.....	283
31	20.4.2 Interface mapping.....	285
32	20.4.3 Interface implementation inheritance.....	287
33	20.4.4 Interface re-implementation	288
34	20.4.5 Abstract classes and interfaces	290
35	21. Enums	291
36	21.1 Enum declarations.....	291
37	21.2 Enum modifiers.....	291
38	21.3 Enum members	292
39	21.4 Enum values and operations	294
40	22. Delegates.....	295
41	22.1 Delegate declarations	295
42	22.2 Delegate instantiation.....	297
43	22.3 Delegate invocation	297
44	23. Exceptions	301
45	23.1 Causes of exceptions.....	301
46	23.2 The <code>System.Exception</code> class.....	301
47	23.3 How exceptions are handled	301
48	23.4 Common Exception Classes	302
49	24. Attributes.....	303
50	24.1 Attribute classes.....	303

C# LANGUAGE SPECIFICATION

1	24.1.1 Attribute usage	303
2	24.1.2 Positional and named parameters	304
3	24.1.3 Attribute parameter types	305
4	24.2 Attribute specification.....	305
5	24.3 Attribute instances	309
6	24.3.1 Compilation of an attribute	309
7	24.3.2 Run-time retrieval of an attribute instance	309
8	24.4 Reserved attributes.....	310
9	24.4.1 The <code>AttributeUsage</code> attribute.....	310
10	24.4.2 The <code>Conditional</code> attribute	310
11	24.4.3 The <code>Obsolete</code> attribute.....	312
12	25. Unsafe code.....	315
13	25.1 Unsafe contexts.....	315
14	25.2 Pointer types	317
15	25.3 Fixed and moveable variables.....	320
16	25.4 Pointer conversions.....	320
17	25.5 Pointers in expressions.....	321
18	25.5.1 Pointer indirection.....	322
19	25.5.2 Pointer member access.....	322
20	25.5.3 Pointer element access	323
21	25.5.4 The address-of operator.....	323
22	25.5.5 Pointer increment and decrement.....	324
23	25.5.6 Pointer arithmetic	324
24	25.5.7 Pointer comparison.....	325
25	25.5.8 The <code>sizeof</code> operator	325
26	25.6 The <code>fixed</code> statement	326
27	25.7 Stack allocation.....	329
28	25.8 Dynamic memory allocation.....	330
29	A. Grammar.....	333
30	A.1 Lexical grammar	333
31	A.1.1 Line terminators	333
32	A.1.2 White space	333
33	A.1.3 Comments.....	333
34	A.1.4 Tokens	334
35	A.1.5 Unicode character escape sequences	334
36	A.1.6 Identifiers	334
37	A.1.7 Keywords	335
38	A.1.8 Literals.....	336
39	A.1.9 Operators and punctuators.....	337
40	A.1.10 Pre-processing directives.....	338
41	A.2 Syntactic grammar	339
42	A.2.1 Basic concepts	339
43	A.2.2 Types	340
44	A.2.3 Variables.....	341
45	A.2.4 Expressions.....	341
46	A.2.5 Statements	344
47	A.2.6 Classes	348
48	A.2.7 Structs.....	353
49	A.2.8 Arrays	354
50	A.2.9 Interfaces	354
51	A.2.10 Enums.....	355
52	A.2.11 Delegates	356
53	A.2.12 Attributes.....	356

1	A.3 Grammar extensions for unsafe code.....	357
2	B. Portability issues	359
3	B.1 Undefined behavior.....	359
4	B.2 Implementation-defined behavior	359
5	B.3 Unspecified behavior.....	360
6	B.4 Other Issues.....	360
7	C. Naming guidelines.....	361
8	C.1 Capitalization styles	361
9	C.1.1 Pascal casing.....	361
10	C.1.2 Camel casing.....	361
11	C.1.3 All uppercase	361
12	C.1.4 Capitalization summary	361
13	C.2 Word choice	362
14	C.3 Namespaces.....	362
15	C.4 Classes.....	362
16	C.5 Interfaces.....	363
17	C.6 Enums.....	363
18	C.7 Static fields.....	364
19	C.8 Parameters	364
20	C.9 Methods.....	364
21	C.10 Properties.....	364
22	C.11 Events.....	365
23	C.12 Case sensitivity.....	365
24	C.13 Avoiding type name confusion	366
25	D. Standard Library.....	367
26	E. Documentation Comments.....	431
27	E.1 Introduction	431
28	E.2 Recommended tags.....	432
29	E.2.1 <c>	432
30	E.2.2 <code>	433
31	E.2.3 <example>	433
32	E.2.4 <exception>	433
33	E.2.5 <list>.....	434
34	E.2.6 <para>.....	435
35	E.2.7 <param>.....	435
36	E.2.8 <paramref>	435
37	E.2.9 <permission>	436
38	E.2.10 <remarks>	436
39	E.2.11 <returns>.....	436
40	E.2.12 <see>.....	437
41	E.2.13 <seealso>	437
42	E.2.14 <summary>	438
43	E.2.15 <value>	438
44	E.3 Processing the documentation file.....	438
45	E.3.1 ID string format	438
46	E.3.2 ID string examples.....	439
47	E.4 An example.....	442
48	E.4.1 C# source code.....	442
49	E.4.2 Resulting XML	444
50	F. Index.....	449
51		

1. Scope

1

2 **This clause is informative.**

3 This ECMA Standard specifies the form and establishes the interpretation of programs written in the
4 C# programming language. It specifies

- 5 • The representation of C# programs;
- 6 • The syntax and constraints of the C# language;
- 7 • The semantic rules for interpreting C# programs;
- 8 • The restrictions and limits imposed by a conforming implementation of C#.

9 This ECMA Standard does not specify

- 10 • The mechanism by which C# programs are transformed for use by a data-processing system;
- 11 • The mechanism by which C# applications are invoked for use by a data-processing system;
- 12 • The mechanism by which input data are transformed for use by a C# application;
- 13 • The mechanism by which output data are transformed after being produced by a C# application;
- 14 • The size or complexity of a program and its data that will exceed the capacity of any specific data-
15 processing system or the capacity of a particular processor;
- 16 • All minimal requirements of a data-processing system that is capable of supporting a conforming
17 implementation.

18 **End of informative text.**

2. Conformance

1

2 Conformance is of interest to the following audiences:

- 3 • Those designing, implementing, or maintaining C# implementations.
- 4 • Governmental or commercial entities wishing to procure C# implementations.
- 5 • Testing organizations wishing to provide a C# conformance test suite.
- 6 • Programmers wishing to port code from one C# implementation to another.
- 7 • Educators wishing to teach Standard C#.
- 8 • Authors wanting to write about Standard C#.

9 As such, conformance is most important, and the bulk of this ECMA Standard is aimed at specifying the
10 characteristics that make C# implementations and C# programs conforming ones.

11

12 The text in this ECMA Standard that specifies requirements is considered *normative*. All other text in this
13 specification is *informative*; that is, for information purposes only. Unless stated otherwise, all text is
14 normative. Normative text is further broken into *required* and *conditional* categories. *Conditionally*
15 *normative* text specifies requirements for a feature such that if that feature is provided, its syntax and
16 semantics must be exactly as specified.

17 If any requirement of this ECMA Standard is violated, the behavior is undefined. Undefined behavior is
18 otherwise indicated in this ECMA Standard by the words “undefined behavior” or by the omission of any
19 explicit definition of behavior. There is no difference in emphasis among these three; they all describe
20 “behavior that is undefined.”

21 A *strictly conforming program* shall use only those features of the language specified in this ECMA
22 Standard as being required. (This means that a strictly conforming program cannot use any conditionally
23 normative feature.) It shall not produce output dependent on any unspecified, undefined, or implementation-
24 defined behavior.

25 A *conforming implementation* of C# must accept any strictly conforming program.

26 A conforming implementation of C# must provide and support all the types, values, objects, properties,
27 methods, and program syntax and semantics described in this ECMA Standard.

28 A conforming implementation of C# shall interpret characters in conformance with the Unicode Standard,
29 Version 3.0 or later, and ISO/IEC 10646-1. Conforming implementations must accept Unicode source files
30 encoded with the UTF-8 encoding form.

31 A conforming implementation of C# shall not successfully translate source containing a `#error`
32 preprocessing directive unless it is part of a group skipped by conditional compilation.

33 A conforming implementation of C# shall produce at least one diagnostic message if the source program
34 violates any rule of syntax, or any negative requirement (defined as a “shall” or “shall not” or “error” or
35 “warning” requirement), unless that requirement is marked with the words “no diagnostic is required”.

36 A conforming implementation of C# is permitted to provide additional types, values, objects, properties, and
37 methods beyond those described in this ECMA Standard, provided they do not alter the behavior of any
38 strictly conforming program. Conforming implementations are required to diagnose programs that use
39 extensions that are ill formed according to this ECMA Standard. Having done so, however; they can compile
40 and execute such programs. (The ability to have extensions implies that a conforming implementation
41 reserves no identifiers other than those explicitly reserved in this ECMA Standard.)

C# LANGUAGE SPECIFICATION

- 1 A conforming implementation of C# shall be accompanied by a document that defines all implementation-
2 defined characteristics, and all extensions.
- 3 A conforming implementation of C# shall support the class library documented in §D. This library is
4 included by reference in this ECMA Standard.
- 5 A *conforming program* is one that is acceptable to a conforming implementation. (Such a program may
6 contain extensions or conditionally normative features.)

3. References

1

2 The following normative documents contain provisions, which, through reference in this text, constitute
3 provisions of this ECMA Standard. For dated references, subsequent amendments to, or revisions of, any of
4 these publications do not apply. However, parties to agreements based on this ECMA Standard are
5 encouraged to investigate the possibility of applying the most recent editions of the normative documents
6 indicated below. For undated references, the latest edition of the normative document referred to applies.
7 Members of ISO and IEC maintain registers of currently valid ECMA Standards.

8

9 ECMA-xxx, 1st Edition, December 2001, *Common Language Infrastructure (CLI), Partition IV: Base Class*
10 *Library (BCL), Extended Numerics Library, and Extended Array Library.*

11 ISO 31.11:1992, *Quantities and units — Part 11: Mathematical signs and symbols for use in the physical*
12 *sciences and technology.*

13 ISO/IEC 2382.1:1993, *Information technology — Vocabulary — Part 1: Fundamental terms.*

14 ISO/IEC 10646 (all parts), *Information technology — Universal Multiple-Octet Coded Character Set (UCS).*

15 IEC 60559:1989, *Binary floating-point arithmetic for microprocessor systems* (previously designated IEC
16 559:1989). (This standard is widely known by its U.S. national designation, ANSI/IEEE Standard 754-1985,
17 *IEEE Standard for Binary Floating-Point Arithmetic.*) Due to the extremely widespread recognition of *IEEE*
18 as the name of a form of floating-point representation and arithmetic, this ECMA Standard uses that term
19 instead of its IEC equivalent.

20 The Unicode Consortium. The Unicode Standard, Version 3.0, defined by: *The Unicode Standard, Version*
21 *3.0* (Reading, MA, Addison-Wesley, 2000. ISBN 0-201-61633-5), and Unicode Technical Report #15:
22 *Unicode Normalization Forms.*

23

24

25 **The following references are informative:**

26

27 ISO/IEC 9899:1999, *Programming languages — C.*

28 ISO/IEC 14882:1998, *Programming languages — C++.*

29 ANSI X3.274-1996, *Programming Language REXX.* (This document is useful in understanding floating-
30 point decimal arithmetic rules.)

31

32 **End of informative references**

4. Definitions

1

2 For the purposes of this ECMA Standard, the following definitions apply. Other terms are defined where
 3 they appear in *italic* type or on the left side of a syntax rule. Terms explicitly defined in this ECMA Standard
 4 are not to be presumed to refer implicitly to similar terms defined elsewhere. Terms not defined in this
 5 ECMA Standard are to be interpreted according to ISO/IEC 2382.1. Mathematical symbols not defined in
 6 this ECMA Standard are to be interpreted according to ISO 31.11.

7

8 **Application** — refers to an assembly that has an entry point (§10.1). When an application is run, a new
 9 application domain is created. Several different instantiations of an application may exist on the same
 10 machine at the same time, and each has its own application domain.

11 **Application domain** — an entity that enables application isolation by acting as a container for application
 12 state. An application domain acts as a container and boundary for the types defined in the application and the
 13 class libraries it uses. Types loaded into one application domain are distinct from the same type loaded into
 14 another application domain, and instances of objects are not directly shared between application domains.
 15 For instance, each application domain has its own copy of static variables for these types, and a static
 16 constructor for a type is run at most once per application domain. Implementations are free to provide
 17 implementation-specific policy or mechanisms for the creation and destruction of application domains.

18 **Argument** — an expression in the comma-separated list bounded by the parentheses in a method or instance
 19 constructor call expression. It is also known as an *actual argument*.

20 **Assembly** — refers to one or more files that are output by the compiler as a result of program compilation.
 21 An assembly is a configured set of loadable code modules and other resources that together implement a unit
 22 of functionality. An assembly may contain types, the executable code used to implement these types, and
 23 references to other assemblies. The physical representation of an assembly is not defined by this
 24 specification. Essentially, an assembly is the output of the compiler.

25 **Behavior** — external appearance or action.

26 **Behavior, implementation-defined** — unspecified behavior where each implementation documents how
 27 the choice is made.

28 **Behavior, undefined** — behavior, upon use of a nonportable or erroneous construct or of erroneous data,
 29 for which this ECMA Standard imposes no requirements. [Possible handling of undefined behavior ranges
 30 from ignoring the situation completely with unpredictable results, to behaving during translation or
 31 execution in a documented manner characteristic of the environment (with or without the issuance of a
 32 diagnostic message), to terminating a translation or execution (with the issuance of a diagnostic message)].

33 **Behavior, unspecified** — behavior where this ECMA Standard provides two or more possibilities and
 34 imposes no further requirements on which is chosen in any instance.

35 **Class library** — refers to an assembly that can be used by other assemblies. Use of a class library does not
 36 cause the creation of a new application domain. Instead, a class library is loaded into the application domain
 37 that uses it. For instance, when an application uses a class library, that class library is loaded into the
 38 application domain for that application. If an application uses a class library A that itself uses a class
 39 library B, then both A and B are loaded into the application domain for the application.

40 **Diagnostic message** — a message belonging to an implementation-defined subset of the implementation's
 41 output messages.

42 **Error, compile-time** — an error reported during program translation.

43 **Exception** — an error condition that is outside the ordinary expected behavior.

C# LANGUAGE SPECIFICATION

1 **Implementation** — particular set of software (running in a particular translation environment under
2 particular control options) that performs translation of programs for, and supports execution of methods in, a
3 particular execution environment.

4 **Namespace** — a logical organizational system that provides a way of presenting program elements that are
5 exposed to other programs.

6 **Parameter** — a variable declared as part of a method, instance constructor, or indexer definition, which
7 acquires a value on entry to that method. It is also known as *formal parameter*.

8 **Program** — refers to one or more source files that are presented to the compiler. Essentially, a program is
9 the input to the compiler.

10 **Program, valid** — a C# program constructed according to the syntax rules and diagnosable semantic rules.

11 **Program instantiation** — the execution of an application.

12 **Recommended practice** — specification that is strongly recommended as being aligned with the intent of
13 the standard, but that may be impractical for some implementations

14 **Source file** — an ordered sequence of Unicode characters. Source files typically have a one-to-one
15 correspondence with files in a file system, but this correspondence is not required.

16 **Unsafe code** — code that is permitted to perform such lower-level operations as declaring and operating on
17 pointers, performing conversions between pointers and integral types, and taking the address of variables.
18 Such operations provide functionality such as permitting interfacing with the underlying operating system,
19 accessing a memory-mapped device, or implementing a time-critical algorithm.

20 **Warning, compile-time** — an informational message reported during program translation, that is intended
21 to identify a potentially questionable usage of a program element.

5. Notational conventions

1

2 Lexical and syntactic grammars for C# are interspersed throughout this specification. The lexical grammar
 3 defines how characters can be combined to form *tokens* (§9.4), the minimal lexical elements of the language.
 4 The syntactic grammar defines how tokens can be combined to make valid C# programs.

5 Grammar productions include both non-terminal and terminal symbols. In grammar productions, *non-*
 6 *terminal* symbols are shown in italic type, and `terminal` symbols are shown in a fixed-width font. Each
 7 non-terminal is defined by a set of productions. The first line of a set of productions is the name of the non-
 8 terminal, followed by a colon. Each successive indented line contains the right-hand side for a production
 9 that has the non-terminal symbol as the left-hand side. For example:

```
10     class-modifier:
11         new
12         public
13         protected
14         internal
15         private
16         abstract
17         sealed
```

18 defines the *class-modifier* non-terminal as having seven productions.

19 Alternatives are normally listed on separate lines, as shown above, though in cases where there are many
 20 alternatives, the phrase “one of” precedes a list of the options. This is simply shorthand for listing each of
 21 the alternatives on a separate line. For example:

```
22     decimal-digit: one of
23         0 1 2 3 4 5 6 7 8 9
```

24 is equivalent to:

```
25     decimal-digit:
26         0
27         1
28         2
29         3
30         4
31         5
32         6
33         7
34         8
35         9
```

36 A subscripted suffix “_{opt}”, as in *identifier_{opt}*, is used as shorthand to indicate an optional symbol. The
 37 example:

```
38     for-statement:
39         for ( for-initializeropt ; for-conditionopt ; for-iteratoropt ) embedded-statement
```

40 is equivalent to:

C# LANGUAGE SPECIFICATION

1 *for-statement*:
2 **for** (; ;) *embedded-statement*
3 **for** (*for-initializer* ; ;) *embedded-statement*
4 **for** (; *for-condition* ;) *embedded-statement*
5 **for** (; ; *for-iterator*) *embedded-statement*
6 **for** (*for-initializer* ; *for-condition* ;) *embedded-statement*
7 **for** (; *for-condition* ; *for-iterator*) *embedded-statement*
8 **for** (*for-initializer* ; ; *for-iterator*) *embedded-statement*
9 **for** (*for-initializer* ; *for-condition* ; *for-iterator*) *embedded-statement*

10

11 All terminal characters are to be understood as the appropriate Unicode character from the ASCII range, as
12 opposed to any similar-looking characters from other Unicode ranges.

6. Acronyms and abbreviations

1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17

This clause is informative.

The following acronyms and abbreviations are used throughout this ECMA Standard:

BCL — Base Class Library, which provides types to represent the built-in data types of the CLI, simple file access, custom attributes, security attributes, string manipulation, formatting, streams, and collections.

CLI — Common Language Infrastructure

CLS — Common Language Specification

IEC — the International Electrotechnical Commission

IEEE — the Institute of Electrical and Electronics Engineers

ISO — the International Organization for Standardization

The name C# is pronounced “C Sharp”.

The name C# is written as the LATIN CAPITAL LETTER C (U+0043) followed by the NUMBER SIGN # (U+000D).

End of informative text.

7. General description

1

2 **This clause is informative.**

3 This ECMA Standard is intended to be used by implementers, academics, and application programmers. As
4 such, it contains a considerable amount of explanatory material that, strictly speaking, is not necessary in a
5 formal language specification.

6 This standard is divided into the following subdivisions:

- 7 1. Front matter (clauses 1–7);
- 8 2. Language overview (clause 8);
- 9 3. The language syntax, constraints, and semantics (clauses 9–25);
- 10 4. Annexes

11 Examples are provided to illustrate possible forms of the constructions described. References are used to
12 refer to related clauses. Notes are provided to give advice or guidance to implementers or programmers.

13 Annexes provide additional information and summarize the information contained in this ECMA Standard.

14 Clauses 2–5, 9–24, the beginning of 25, and the beginning of D form a normative part of this standard; all of
15 clause 25 with the exception of the beginning is conditionally normative; and Brief history, clauses 1, 6–8,
16 annexes A, B, C, and most of D, notes, examples, and the index are informative.

17 Except for whole clauses or annexes that are identified as being informative, informative text that is
18 contained within normative text is indicated in two ways:

- 19 1. [*Example*: The following example ... code fragment, possibly with some narrative ... *end example*]
- 20 2. [*Note*: narrative ... *end note*]

21 **End of informative text.**

8. Language Overview

1

2 **This clause is informative.**

3 C# (pronounced “C Sharp”) is a simple, modern, object oriented, and type-safe programming language. It
 4 will immediately be familiar to C and C++ programmers. C# combines the high productivity of Rapid
 5 Application Development (RAD) languages and the raw power of C++.

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

10 **8.1 Getting started**

11 The canonical “hello, world” program can be written as follows:

```
12     using System;
13     class Hello
14     {
15         static void Main() {
16             Console.WriteLine("hello, world");
17         }
18     }
```

19 The source code for a C# program is typically stored in one or more text files with a file extension of `.cs`, as
 20 in `hello.cs`. Using a command-line compiler, such a program can be compiled with a command line like

```
21     csc hello.cs
```

22 which produces an application named `hello.exe`. The output produced by this application when it is run
 23 is:

```
24     hello, world
```

25 Close examination of this program is illuminating:

- 26 • The `using System;` directive references a namespace called `System` that is provided by the Common
 27 Language Infrastructure (CLI) class library. This namespace contains the `Console` class referred to in
 28 the `Main` method. Namespaces provide a hierarchical means of organizing the elements of one or more
 29 programs. A `using`-directive enables unqualified use of the types that are members of the namespace.
 30 The “hello, world” program uses `Console.WriteLine` as shorthand for
 31 `System.Console.WriteLine`.
- 32 • The `Main` method is a member of the class `Hello`. It has the `static` modifier, and so it is a method on
 33 the class `Hello` rather than on instances of this class.
- 34 • The entry point for an application—the method that is called to begin execution—is always a static
 35 method named `Main`.
- 36 • The “hello, world” output is produced using a class library. This standard does not include a class
 37 library. Instead, it references the class library provided by CLI.

38 For C and C++ developers, it is interesting to note a few things that do *not* appear in the “hello, world”
 39 program.

- 40 • The program does not use a global method for `Main`. Methods and variables are not supported at the
 41 global level; such elements are always contained within type declarations (e.g., class and struct
 42 declarations).

- 1 • The program does not use either “:” or “->” operators. The “:” is not an operator at all, and the
2 “->” operator is used in only a small fraction of programs (which involve unsafe code). The
3 separator “.” is used in compound names such as `Console.WriteLine`.
- 4 • The program does not contain forward declarations. Forward declarations are never needed, as
5 declaration order is not significant.
- 6 • The program does not use `#include` to import program text. Dependencies among programs are
7 handled symbolically rather than textually. This approach eliminates barriers between applications
8 written using multiple languages. For example, the `Console` class need not be written in C#.

9 8.2 Types

10 C# supports two kinds of types: *value types* and *reference types*. Value types include simple types (e.g.,
11 `char`, `int`, and `float`), enum types, and struct types. Reference types include class types, interface types,
12 delegate types, and array types.

13 Value types differ from reference types in that variables of the value types directly contain their data,
14 whereas variables of the reference types store references to objects. With reference types, it is possible for
15 two variables to reference the same object, and thus possible for operations on one variable to affect the
16 object referenced by the other variable. With value types, the variables each have their own copy of the data,
17 and it is not possible for operations on one to affect the other.

18 The example

```
19     using System;
20     class Class1
21     {
22         public int value = 0;
23     }
24     class Test
25     {
26         static void Main() {
27             int val1 = 0;
28             int val2 = val1;
29             val2 = 123;
30             Class1 ref1 = new Class1();
31             Class1 ref2 = ref1;
32             ref2.Value = 123;
33             Console.WriteLine("Values: {0}, {1}", val1, val2);
34             Console.WriteLine("Refs: {0}, {1}", ref1.Value, ref2.Value);
35         }
36     }
```

37 shows this difference. The output produced is

```
38     Values: 0, 123
39     Refs: 123, 123
```

40 The assignment to the local variable `val1` does not impact the local variable `val2` because both local
41 variables are of a value type (the type `int`) and each local variable of a value type has its own storage. In
42 contrast, the assignment `ref2.Value = 123`; affects the object that both `ref1` and `ref2` reference.

43 The lines

```
44     Console.WriteLine("Values: {0}, {1}", val1, val2);
45     Console.WriteLine("Refs: {0}, {1}", ref1.Value, ref2.Value);
```

46 deserve further comment, as they demonstrate some of the string formatting behavior of
47 `Console.WriteLine`, which, in fact, takes a variable number of arguments. The first argument is a string,
48 which may contain numbered placeholders like `{0}` and `{1}`. Each placeholder refers to a trailing argument
49 with `{0}` referring to the second argument, `{1}` referring to the third argument, and so on. Before the output
50 is sent to the console, each placeholder is replaced with the formatted value of its corresponding argument.

1 Developers can define new value types through enum and struct declarations, and can define new reference
2 types via class, interface, and delegate declarations. The example

```

3     using System;
4     public enum Color
5     {
6         Red, Blue, Green
7     }
8     public struct Point
9     {
10        public int x, y;
11    }
12    public interface IBase
13    {
14        void F();
15    }
16    public interface IDerived: IBase
17    {
18        void G();
19    }
20    public class A
21    {
22        protected virtual void H() {
23            Console.WriteLine("A.H");
24        }
25    }
26    public class B: A, IDerived
27    {
28        public void F() {
29            Console.WriteLine("B.F, implementation of IDerived.F");
30        }
31        public void G() {
32            Console.WriteLine("B.G, implementation of IDerived.G");
33        }
34        override protected void H() {
35            Console.WriteLine("B.H, override of A.H");
36        }
37    }
38    public delegate void EmptyDelegate();

```

39 shows an example of each kind of type declaration. Later sections describe type declarations in detail.

40 8.2.1 Predefined types

41 C# provides a set of predefined types, most of which will be familiar to C and C++ developers.

42 The predefined reference types are `object` and `string`. The type `object` is the ultimate base type of all
43 other types. The type `string` is used to represent Unicode string values. Values of type `string` are
44 immutable.

45 The predefined value types include signed and unsigned integral types, floating-point types, and the types
46 `bool`, `char`, and `decimal`. The signed integral types are `sbyte`, `short`, `int`, and `long`; the unsigned
47 integral types are `byte`, `ushort`, `uint`, and `ulong`; and the floating-point types are `float` and `double`.

48 The `bool` type is used to represent boolean values: values that are either true or false. The inclusion of `bool`
49 makes it easier to write self-documenting code, and also helps eliminate the all-too-common C++ coding
50 error in which a developer mistakenly uses “=” when “==” should have been used. In C#, the example

```

51     int i = ...;
52     F(i);
53     if (i = 0) // Bug: the test should be (i == 0)
54         G();

```

C# LANGUAGE SPECIFICATION

1 results in a compile-time error because the expression `i = 0` is of type `int`, and `if` statements require an
2 expression of type `bool`.

3 The `char` type is used to represent Unicode characters. A variable of type `char` represents a single 16-bit
4 Unicode character.

5 The `decimal` type is appropriate for calculations in which rounding errors caused by floating point
6 representations are unacceptable. Common examples include financial calculations such as tax computations
7 and currency conversions. The `decimal` type provides 28 significant digits.

8 The table below lists the predefined types, and shows how to write literal values for each of them.

9

Type	Description	Example
<code>object</code>	The ultimate base type of all other types	<code>object o = null;</code>
<code>string</code>	String type; a string is a sequence of Unicode characters	<code>string s = "hello";</code>
<code>sbyte</code>	8-bit signed integral type	<code>sbyte val = 12;</code>
<code>short</code>	16-bit signed integral type	<code>short val = 12;</code>
<code>int</code>	32-bit signed integral type	<code>int val = 12;</code>
<code>long</code>	64-bit signed integral type	<code>long val1 = 12;</code> <code>long val2 = 34L;</code>
<code>byte</code>	8-bit unsigned integral type	<code>byte val1 = 12;</code>
<code>ushort</code>	16-bit unsigned integral type	<code>ushort val1 = 12;</code>
<code>uint</code>	32-bit unsigned integral type	<code>uint val1 = 12;</code> <code>uint val2 = 34U;</code>
<code>ulong</code>	64-bit unsigned integral type	<code>ulong val1 = 12;</code> <code>ulong val2 = 34U;</code> <code>ulong val3 = 56L;</code> <code>ulong val4 = 78UL;</code>
<code>float</code>	Single-precision floating point type	<code>float val = 1.23F;</code>
<code>double</code>	Double-precision floating point type	<code>double val1 = 1.23;</code> <code>double val2 = 4.56D;</code>
<code>bool</code>	Boolean type; a <code>bool</code> value is either true or false	<code>bool val1 = true;</code> <code>bool val2 = false;</code>
<code>char</code>	Character type; a <code>char</code> value is a Unicode character	<code>char val = 'h';</code>
<code>decimal</code>	Precise decimal type with 28 significant digits	<code>decimal val = 1.23M;</code>

10

11 Each of the predefined types is shorthand for a system-provided type. For example, the keyword `int` refers
12 to the struct `System.Int32`. As a matter of style, use of the keyword is favored over use of the complete
13 system type name.

14 Predefined value types such as `int` are treated specially in a few ways but are for the most part treated
15 exactly like other structs. Operator overloading enables developers to define new struct types that behave
16 much like the predefined value types. For instance, a `Digit` struct can support the same mathematical
17 operations as the predefined integral types, and can define conversions between `Digit` and predefined
18 types.

19 The predefined types employ operator overloading themselves. For example, the comparison operators `==`
20 and `!=` have different semantics for different predefined types:

- 21 • Two expressions of type `int` are considered equal if they represent the same integer value.

- 1 • Two expressions of type `object` are considered equal if both refer to the same object, or if both are
2 `null`.
- 3 • Two expressions of type `string` are considered equal if the string instances have identical lengths and
4 identical characters in each character position, or if both are `null`.

5 The example

```
6     using System;
7     class Test
8     {
9         static void Main() {
10            string s = "Test";
11            string t = string.Copy(s);
12            Console.WriteLine(s == t);
13            Console.WriteLine((object)s == (object)t);
14        }
15    }
```

16 produces the output

```
17     True
18     False
```

19 because the first comparison compares two expressions of type `string`, and the second comparison
20 compares two expressions of type `object`.

21 8.2.2 Conversions

22 The predefined types also have predefined conversions. For instance, conversions exist between the
23 predefined types `int` and `long`. C# differentiates between two kinds of conversions: *implicit conversions*
24 and *explicit conversions*. Implicit conversions are supplied for conversions that can safely be performed
25 without careful scrutiny. For instance, the conversion from `int` to `long` is an implicit conversion. This
26 conversion always succeeds, and never results in a loss of information. The following example

```
27     using System;
28     class Test
29     {
30         static void Main() {
31            int intValue = 123;
32            long longValue = intValue;
33            Console.WriteLine("{0}, {1}", intValue, longValue);
34        }
35    }
```

36 implicitly converts an `int` to a `long`.

37 In contrast, explicit conversions are performed with a cast expression. The example

```
38     using System;
39     class Test
40     {
41         static void Main() {
42            long longValue = Int64.MaxValue;
43            int intValue = (int) longValue;
44            Console.WriteLine("(int) {0} = {1}", longValue, intValue);
45        }
46    }
```

47 uses an explicit conversion to convert a `long` to an `int`. The output is:

```
48     (int) 9223372036854775807 = -1
```

49 because an overflow occurs. Cast expressions permit the use of both implicit and explicit conversions.

50 8.2.3 Array types

51 Arrays may be single-dimensional or multi-dimensional. Both “rectangular” and “jagged” arrays are
52 supported.

1 Single-dimensional arrays are the most common type. The example

```

2     using System;
3     class Test
4     {
5         static void Main() {
6             int[] arr = new int[5];
7             for (int i = 0; i < arr.Length; i++)
8                 arr[i] = i * i;
9             for (int i = 0; i < arr.Length; i++)
10                Console.WriteLine("arr[{0}] = {1}", i, arr[i]);
11        }
12    }

```

13 creates a single-dimensional array of `int` values, initializes the array elements, and then prints each of them
14 out. The output produced is:

```

15     arr[0] = 0
16     arr[1] = 1
17     arr[2] = 4
18     arr[3] = 9
19     arr[4] = 16

```

20 The type `int[]` used in the previous example is an array type. Array types are written using a non-array-
21 type followed by one or more rank specifiers. The example

```

22     class Test
23     {
24         static void Main() {
25             int[] a1;           // single-dimensional array of int
26             int[,] a2;         // 2-dimensional array of int
27             int[,,] a3;        // 3-dimensional array of int
28             int[][] j2;        // "jagged" array: array of (array of int)
29             int[][][] j3;      // array of (array of (array of int))
30         }
31     }

```

32 shows a variety of local variable declarations that use array types with `int` as the element type.

33 Array types are reference types, and so the declaration of an array variable merely sets aside space for the
34 reference to the array. Array instances are actually created via array initializers and array creation
35 expressions. The example

```

36     class Test
37     {
38         static void Main() {
39             int[] a1 = new int[] {1, 2, 3};
40             int[,] a2 = new int[,] {{1, 2, 3}, {4, 5, 6}};
41             int[,,] a3 = new int[10, 20, 30];
42             int[][] j2 = new int[3][];
43             j2[0] = new int[] {1, 2, 3};
44             j2[1] = new int[] {1, 2, 3, 4, 5, 6};
45             j2[2] = new int[] {1, 2, 3, 4, 5, 6, 7, 8, 9};
46         }
47     }

```

48 shows a variety of array creation expressions. The variables `a1`, `a2` and `a3` denote *rectangular arrays*, and
49 the variable `j2` denotes a *jagged array*. It should be no surprise that these terms are based on the shapes of
50 the arrays. Rectangular arrays always have a rectangular shape. Given the length of each dimension of the
51 array, its rectangular shape is clear. For example, the lengths of `a3`'s three dimensions are 10, 20, and 30,
52 respectively, and it is easy to see that this array contains $10 \times 20 \times 30$ elements.

53 In contrast, the variable `j2` denotes a "jagged" array, or an "array of arrays". Specifically, `j2` denotes an
54 array of an array of `int`, or a single-dimensional array of type `int[]`. Each of these `int[]` variables can be
55 initialized individually, and this allows the array to take on a jagged shape. The example gives each of the

1 `int[]` arrays a different length. Specifically, the length of `j2[0]` is 3, the length of `j2[1]` is 6, and the
2 length of `j2[2]` is 9.

3 [Note: In C++, an array declared as `int x[3][5][7]` would be considered a three dimensional rectangular
4 array, while in C#, the declaration `int[][][]` declares a jagged array type. *end note*]

5 The element type and shape of an array—including whether it is jagged or rectangular, and the number of
6 dimensions it has—are part of its type. On the other hand, the size of the array—as represented by the length
7 of each of its dimensions—is not part of an array’s type. This split is made clear in the language syntax, as
8 the length of each dimension is specified in the array creation expression rather than in the array type. For
9 instance the declaration

```
10     int[, ,] a3 = new int[10, 20, 30];
```

11 has an array type of `int[, ,]` and an array creation expression of `new int[10, 20, 30]`.

12 For local variable and field declarations, a shorthand form is permitted so that it is not necessary to re-state
13 the array type. For instance, the example

```
14     int[] a1 = new int[] {1, 2, 3};
```

15 can be shortened to

```
16     int[] a1 = {1, 2, 3};
```

17 without any change in program semantics.

18 The context in which an array initializer such as `{1, 2, 3}` is used determines the type of the array being
19 initialized. The example

```
20     class Test
21     {
22         static void Main() {
23             short[] a = {1, 2, 3};
24             int[] b = {1, 2, 3};
25             long[] c = {1, 2, 3};
26         }
27     }
```

28 shows that the same array initializer syntax can be used for several different array types. Because context is
29 required to determine the type of an array initializer, it is not possible to use an array initializer in an
30 expression context without explicitly stating the type of the array.

31 8.2.4 Type system unification

32 C# provides a “unified type system”. All types—including value types—derive from the type `object`. It is
33 possible to call object methods on any value, even values of “primitive” types such as `int`. The example

```
34     using System;
35     class Test
36     {
37         static void Main() {
38             Console.WriteLine(3.ToString());
39         }
40     }
```

41 calls the object-defined `ToString` method on an integer literal, resulting in the output “3”.

42 The example

```
43     class Test
44     {
45         static void Main() {
46             int i = 123;
47             object o = i; // boxing
48             int j = (int) o; // unboxing
49         }
50     }
```

1 is more interesting. An `int` value can be converted to `object` and back again to `int`. This example shows
 2 both *boxing* and *unboxing*. When a variable of a value type needs to be converted to a reference type, an
 3 object *box* is allocated to hold the value, and the value is copied into the box. *Unboxing* is just the opposite.
 4 When an object box is cast back to its original value type, the value is copied out of the box and into the
 5 appropriate storage location.

6 This type system unification provides value types with the benefits of object-ness without introducing
 7 unnecessary overhead. For programs that don't need `int` values to act like objects, `int` values are simply
 8 32-bit values. For programs that need `int` values to behave like objects, this capability is available on
 9 demand. This ability to treat value types as objects bridges the gap between value types and reference types
 10 that exists in most languages. For example, a `Stack` class can provide `Push` and `Pop` methods that take and
 11 return `object` values.

```
12     public class Stack
13     {
14         public object Pop() {...}
15         public void Push(object o) {...}
16     }
```

17 Because C# has a unified type system, the `Stack` class can be used with elements of any type, including
 18 value types like `int`.

19 8.3 Variables and parameters

20 *Variables* represent storage locations. Every variable has a type that determines what values can be stored in
 21 the variable. *Local variables* are variables that are declared in methods, properties, or indexers. A local
 22 variable is defined by specifying a type name and a declarator that specifies the variable name and an
 23 optional initial value, as in:

```
24     int a;
25     int b = 1;
```

26 but it is also possible for a local variable declaration to include multiple declarators. The declarations of `a`
 27 and `b` can be rewritten as:

```
28     int a, b = 1;
```

29 A variable must be assigned before its value can be obtained. The example

```
30     class Test
31     {
32         static void Main() {
33             int a;
34             int b = 1;
35             int c = a + b; // error, a not yet assigned
36             ...
37         }
38     }
```

39 results in a compile-time error because it attempts to use the variable `a` before it is assigned a value. The
 40 rules governing definite assignment are defined in §12.3.

41 A *field* (§17.4) is a variable that is associated with a class or struct, or an instance of a class or struct. A field
 42 declared with the `static` modifier defines a *static variable*, and a field declared without this modifier
 43 defines an *instance variable*. A static field is associated with a type, whereas an instance variable is
 44 associated with an instance. The example

```
45     using Personnel.Data;
46     class Employee
47     {
48         private static DataSet ds;
49         public string Name;
50         public decimal Salary;
51         ...
52     }
```

1 shows an `Employee` class that has a private static variable and two public instance variables.

2 Formal parameter declarations also define variables. There are four kinds of parameters: value parameters,
3 reference parameters, output parameters, and parameter arrays.

4 A *value parameter* is used for “in” parameter passing, in which the value of an argument is passed into a
5 method, and modifications of the parameter do not impact the original argument. A value parameter refers to
6 its own variable, one that is distinct from the corresponding argument. This variable is initialized by copying
7 the value of the corresponding argument. The example

```
8     using System;
9     class Test {
10         static void F(int p) {
11             Console.WriteLine("p = {0}", p);
12             p++;
13         }
14         static void Main() {
15             int a = 1;
16             Console.WriteLine("pre: a = {0}", a);
17             F(a);
18             Console.WriteLine("post: a = {0}", a);
19         }
20     }
```

21 shows a method `F` that has a value parameter named `p`. The example produces the output:

```
22     pre: a = 1
23     p = 1
24     post: a = 1
```

25 even though the value parameter `p` is modified.

26 A *reference parameter* is used for “by reference” parameter passing, in which the parameter acts as an alias
27 for a caller-provided argument. A reference parameter does not itself define a variable, but rather refers to
28 the variable of the corresponding argument. Modifications of a reference parameter impact the
29 corresponding argument. A reference parameter is declared with a `ref` modifier. The example

```
30     using System;
31     class Test {
32         static void Swap(ref int a, ref int b) {
33             int t = a;
34             a = b;
35             b = t;
36         }
37         static void Main() {
38             int x = 1;
39             int y = 2;
40
41             Console.WriteLine("pre: x = {0}, y = {1}", x, y);
42             Swap(ref x, ref y);
43             Console.WriteLine("post: x = {0}, y = {1}", x, y);
44         }
45     }
```

46 shows a `Swap` method that has two reference parameters. The output produced is:

```
47     pre: x = 1, y = 2
48     post: x = 2, y = 1
```

49 The `ref` keyword must be used in both the declaration of the formal parameter and in uses of it. The use of
50 `ref` at the call site calls special attention to the parameter, so that a developer reading the code will
51 understand that the value of the argument could change as a result of the call.

52 An *output parameter* is similar to a reference parameter, except that the initial value of the caller-provided
53 argument is unimportant. An output parameter is declared with an `out` modifier. The example

C# LANGUAGE SPECIFICATION

```
1     using System;
2     class Test {
3         static void Divide(int a, int b, out int result, out int remainder) {
4             result = a / b;
5             remainder = a % b;
6         }
7
8         static void Main() {
9             for (int i = 1; i < 10; i++)
10                for (int j = 1; j < 10; j++) {
11                    int ans, r;
12                    Divide(i, j, out ans, out r);
13                    Console.WriteLine("{0} / {1} = {2}r{3}", i, j, ans, r);
14                }
15        }
16    }
```

16 shows a `Divide` method that includes two output parameters—one for the result of the division and another for the remainder.

18 For value, reference, and output parameters, there is a one-to-one correspondence between caller-provided arguments and the parameters used to represent them. A *parameter array* enables a many-to-one relationship: many arguments can be represented by a single parameter array. In other words, parameter arrays enable variable length argument lists.

22 A parameter array is declared with a `params` modifier. There can be only one parameter array for a given method, and it must always be the last parameter specified. The type of a parameter array is always a single dimensional array type. A caller can either pass a single argument of this array type, or any number of arguments of the element type of this array type. For instance, the example

```
26     using System;
27     class Test
28     {
29         static void F(params int[] args) {
30             Console.WriteLine("# of arguments: {0}", args.Length);
31             for (int i = 0; i < args.Length; i++)
32                 Console.WriteLine("\targs[{0}] = {1}", i, args[i]);
33         }
34
35         static void Main() {
36             F();
37             F(1);
38             F(1, 2);
39             F(1, 2, 3);
40             F(new int[] {1, 2, 3, 4});
41         }
42     }
```

42 shows a method `F` that takes a variable number of `int` arguments, and several invocations of this method. The output is:

```
44     # of arguments: 0
45     # of arguments: 1
46     args[0] = 1
47     # of arguments: 2
48     args[0] = 1
49     args[1] = 2
50     # of arguments: 3
51     args[0] = 1
52     args[1] = 2
53     args[2] = 3
54     # of arguments: 4
55     args[0] = 1
56     args[1] = 2
57     args[2] = 3
58     args[3] = 4
```

59 Most of the examples presented in this introduction use the `writeLine` method of the `Console` class. The argument substitution behavior of this method, as exhibited in the example


```

1      int a = 1, b = 2;
2      Console.WriteLine("a = {0}, b = {1}", a, b);

```

3 is accomplished using a parameter array. The `WriteLine` method provides several overloaded methods for
4 the common cases in which a small number of arguments are passed, and one method that uses a parameter
5 array.

```

6      namespace System
7      {
8          public class Console
9          {
10             public static void WriteLine(string s) {...}
11             public static void WriteLine(string s, object a) {...}
12             public static void WriteLine(string s, object a, object b) {...}
13             ...
14             public static void WriteLine(string s, params object[] args) {...}
15         }
16     }

```

17 8.4 Automatic memory management

18 *Manual memory management* requires developers to manage the allocation and de-allocation of blocks of
19 memory. Manual memory management can be both time-consuming and difficult. In C#, *automatic memory*
20 *management* is provided so that developers are freed from this burdensome task. In the vast majority of
21 cases, automatic memory management increases code quality and enhances developer productivity without
22 negatively impacting either expressiveness or performance.

23 The example

```

24     using System;
25     public class Stack
26     {
27         private Node first = null;
28
29         public bool Empty {
30             get {
31                 return (first == null);
32             }
33         }
34         public object Pop() {
35             if (first == null)
36                 throw new Exception("Can't Pop from an empty Stack.");
37             else {
38                 object temp = first.Value;
39                 first = first.Next;
40                 return temp;
41             }
42         }
43         public void Push(object o) {
44             first = new Node(o, first);
45         }
46         class Node
47         {
48             public Node Next;
49             public object Value;
50             public Node(object value): this(value, null) {}
51             public Node(object value, Node next) {
52                 Next = next;
53                 Value = value;
54             }
55         }
56     }

```

56 shows a `Stack` class implemented as a linked list of `Node` instances. `Node` instances are created in the `Push`
57 method and are garbage collected when no longer needed. A `Node` instance becomes eligible for garbage

C# LANGUAGE SPECIFICATION

1 collection when it is no longer possible for any code to access it. For instance, when an item is removed
2 from the `Stack`, the associated `Node` instance becomes eligible for garbage collection.

3 The example

```
4     class Test
5     {
6         static void Main() {
7             Stack s = new Stack();
8             for (int i = 0; i < 10; i++)
9                 s.Push(i);
10            s = null;
11        }
12    }
```

13 shows code that uses the `Stack` class. A `Stack` is created and initialized with 10 elements, and then
14 assigned the value `null`. Once the variable `s` is assigned `null`, the `Stack` and the associated 10 `Node`
15 instances become eligible for garbage collection. The garbage collector is permitted to clean up immediately,
16 but is not required to do so.

17 The garbage collector underlying C# may work by moving objects around in memory, but this motion is
18 invisible to most C# developers. For developers who are generally content with automatic memory
19 management but sometimes need fine-grained control or that extra bit of performance, C# provides the
20 ability to write “unsafe” code. Such code can deal directly with pointer types and object addresses, however,
21 C# requires the programmer to *fix* objects to temporarily prevent the garbage collector from moving them.

22 This “unsafe” code feature is in fact a “safe” feature from the perspective of both developers and users.
23 Unsafe code must be clearly marked in the code with the modifier `unsafe`, so developers can't possibly use
24 unsafe language features accidentally, and the compiler and the execution engine work together to ensure
25 that unsafe code cannot masquerade as safe code. These restrictions limit the use of unsafe code to situations
26 in which the code is trusted.

27 The example

```
28     using System;
29     class Test
30     {
31         static void writeLocations(byte[] arr) {
32             unsafe {
33                 fixed (byte* pArray = arr) {
34                     byte* pElem = pArray;
35                     for (int i = 0; i < arr.Length; i++) {
36                         byte value = *pElem;
37                         Console.WriteLine("arr[{0}] at 0x{1:X} is {2}",
38                             i, (uint)pElem, value);
39                         pElem++;
40                     }
41                 }
42             }
43         }
44         static void Main() {
45             byte[] arr = new byte[] {1, 2, 3, 4, 5};
46             writeLocations(arr);
47         }
48     }
```

49 shows an unsafe block in a method named `writeLocations` that fixes an array instance and uses pointer
50 manipulation to iterate over the elements. The index, value, and location of each array element are written to
51 the console. One possible example of output is:

```
52     arr[0] at 0x8E0360 is 1
53     arr[1] at 0x8E0361 is 2
54     arr[2] at 0x8E0362 is 3
55     arr[3] at 0x8E0363 is 4
56     arr[4] at 0x8E0364 is 5
```

57 but, of course, the exact memory locations may be different in different executions of the application.

8.5 Expressions

C# includes unary operators, binary operators, and one ternary operator. The following table summarizes the operators, listing them in order of precedence from highest to lowest:

Section	Category	Operators
14.5	Primary	x.y f(x) a[x] x++ x-- new typeof checked unchecked
0	Unary	+ - ! ~ ++x --x (T)x
14.7	Multiplicative	* / %
14.7	Additive	+ -
0	Shift	<< >>
14.9	Relational and type-testing	< > <= >= is as
14.9	Equality	== !=
14.10	Logical AND	&
14.10	Logical XOR	^
14.10	Logical OR	
14.11	Conditional AND	&&
14.11	Conditional OR	
14.12	Conditional	?:
14.13	Assignment	= *= /= %= += -= <<= >>= &= ^= =

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.

When an operand occurs between two operators with the same precedence, the *associativity* of the operators controls the order in which the operations are performed:

- Except for the assignment operators, all binary operators are *left-associative*, meaning that operations are performed from left to right. For example, `x + y + z` is evaluated as `(x + y) + z`.
- The assignment operators and the conditional operator (`?:`) are *right-associative*, meaning that operations are performed from right to left. For example, `x = y = z` is evaluated as `x = (y = z)`.

Precedence and associativity can be controlled using parentheses. For example, `x + y * z` first multiplies `y` by `z` and then adds the result to `x`, but `(x + y) * z` first adds `x` and `y` and then multiplies the result by `z`.

8.6 Statements

C# borrows most of its statements directly from C and C++, though there are some noteworthy additions and modifications. The table below lists the kinds of statements that can be used, and provides an example for each.

Statement	Example
Statement lists and block statements	<pre>static void Main() { F(); G(); { H(); I(); } }</pre>
Labeled statements and goto statements	<pre>static void Main(string[] args) { if (args.Length == 0) goto done; Console.WriteLine(args.Length); done: Console.WriteLine("Done"); }</pre>
Local constant declarations	<pre>static void Main() { const float pi = 3.14f; const int r = 123; Console.WriteLine(pi * r * r); }</pre>
Local variable declarations	<pre>static void Main() { int a; int b = 2, c = 3; a = 1; Console.WriteLine(a + b + c); }</pre>
Expression statements	<pre>static int F(int a, int b) { return a + b; } static void Main() { F(1, 2); // Expression statement }</pre>
if statements	<pre>static void Main(string[] args) { if (args.Length == 0) Console.WriteLine("No args"); else Console.WriteLine("Args"); }</pre>
switch statements	<pre>static void Main(string[] args) { switch (args.Length) { case 0: Console.WriteLine("No args"); break; case 1: Console.WriteLine("One arg "); break; default: int n = args.Length; Console.WriteLine("{0} args", n); break; } }</pre>
while statements	<pre>static void Main(string[] args) { int i = 0; while (i < args.Length) { Console.WriteLine(args[i]); i++; } }</pre>

do statements	<pre>static void Main() { string s; do { s = Console.ReadLine(); } while (s != "Exit"); }</pre>
for statements	<pre>static void Main(string[] args) { for (int i = 0; i < args.Length; i++) Console.WriteLine(args[i]); }</pre>
foreach statements	<pre>static void Main(string[] args) { foreach (string s in args) Console.WriteLine(s); }</pre>
break statements	<pre>static void Main(string[] args) { int i = 0; while (true) { if (i == args.Length) break; Console.WriteLine(args[i++]); } }</pre>
continue statements	<pre>static void Main(string[] args) { int i = 0; while (true) { Console.WriteLine(args[i++]); if (i < args.Length) continue; break; } }</pre>
return statements	<pre>static int F(int a, int b) { return a + b; } static void Main() { Console.WriteLine(F(1, 2)); return; }</pre>
throw statements and try statements	<pre>static int F(int a, int b) { if (b == 0) throw new Exception("Divide by zero"); return a / b; } static void Main() { try { Console.WriteLine(F(5, 0)); } catch (Exception e) { Console.WriteLine("Error"); } }</pre>
checked and unchecked statements	<pre>static void Main() { int x = Int32.MaxValue; Console.WriteLine(x + 1); // Overflow checked { Console.WriteLine(x + 1); // Exception } unchecked { Console.WriteLine(x + 1); // Overflow } }</pre>

lock statements	<pre>static void Main() { A a = ...; lock(a) { a.P = a.P + 1; } }</pre>
using statements	<pre>static void Main() { using (Resource r = new Resource()) { r.F(); } }</pre>

1
2
3
4
5
6
7
8
9

8.7 Classes

Class declarations define new reference types. A class can inherit from another class, and can implement interfaces.

Class members can include constants, fields, methods, properties, events, indexers, operators, instance constructors, destructors, static constructors, and nested type declarations. Each member has an associated accessibility (§10.5), which controls the regions of program text that are able to access the member. There are five possible forms of accessibility. These are summarized in the table below.

Form	Intuitive meaning
public	Access not limited
protected	Access limited to the containing class or types derived from the containing class
internal	Access limited to this program
protected internal	Access limited to this program or types derived from the containing class
private	Access limited to the containing type

10
11
12
13
14
15
16
17
18
19
20
21
22
23
24
25
26
27
28
29
30
31
32
33

The example

```
using System;
class MyClass
{
    public MyClass() {
        Console.WriteLine("Instance constructor");
    }

    public MyClass(int value) {
        MyField = value;
        Console.WriteLine("Instance constructor");
    }

    ~MyClass() {
        Console.WriteLine("Destructor");
    }

    public const int MyConst = 12;
    public int MyField = 34;
    public void MyMethod(){
        Console.WriteLine("MyClass.MyMethod");
    }

    public int MyProperty {
        get {
            return MyField;
        }
    }
}
```

```

1         set {
2             MyField = value;
3         }
4     }
5     public int this[int index] {
6         get {
7             return 0;
8         }
9         set {
10            Console.WriteLine("this[{0}] = {1}", index, value);
11        }
12    }
13    public event EventHandler MyEvent;
14    public static MyClass operator+(MyClass a, MyClass b) {
15        return new MyClass(a.MyField + b.MyField);
16    }
17    internal class MyNestedClass
18    {}
19 }

```

20 shows a class that contains each kind of member. The example

```

21 class Test
22 {
23     static void Main() {
24         // Instance constructor usage
25         MyClass a = new MyClass();
26         MyClass b = new MyClass(123);
27         // Constant usage
28         Console.WriteLine("MyConst = {0}", MyClass.MyConst);
29         // Field usage
30         a.MyField++;
31         Console.WriteLine("a.MyField = {0}", a.MyField);
32         // Method usage
33         a.MyMethod();
34         // Property usage
35         a.MyProperty++;
36         Console.WriteLine("a.MyProperty = {0}", a.MyProperty);
37         // Indexer usage
38         a[3] = a[1] = a[2];
39         Console.WriteLine("a[3] = {0}", a[3]);
40         // Event usage
41         a.MyEvent += new EventHandler(MyHandler);
42         // Overloaded operator usage
43         MyClass c = a + b;
44     }
45     static void MyHandler(object sender, EventArgs e) {
46         Console.WriteLine("Test.MyHandler");
47     }
48     internal class MyNestedClass
49     {}
50 }

```

51 shows uses of these members.

52 8.7.1 Constants

53 A *constant* is a class member that represents a constant value: a value that can be computed at compile-time.
54 Constants are permitted to depend on other constants within the same program as long as there are no
55 circular dependencies. The rules governing constant expressions are defined in §14.15. The example

```

1      class Constants
2      {
3          public const int A = 1;
4          public const int B = A + 1;
5      }

```

6 shows a class named `Constants` that has two public constants.

7 Even though constants are considered static members, a constant declaration neither requires nor allows the
8 modifier `static`. Constants can be accessed through the class, as in

```

9      using System;
10     class Test
11     {
12         static void Main() {
13             Console.WriteLine("{0}, {1}", Constants.A, Constants.B);
14         }
15     }

```

16 which prints out the values of `Constants.A` and `Constants.B`, respectively.

17 8.7.2 Fields

18 A *field* is a member that represents a variable associated with an object or class. The example

```

19     class Color
20     {
21         internal ushort redPart;
22         internal ushort bluePart;
23         internal ushort greenPart;
24
25         public Color(ushort red, ushort blue, ushort green) {
26             redPart = red;
27             bluePart = blue;
28             greenPart = green;
29
30             public static Color Red = new Color(0xFF, 0, 0);
31             public static Color Blue = new Color(0, 0xFF, 0);
32             public static Color Green = new Color(0, 0, 0xFF);
33             public static Color White = new Color(0xFF, 0xFF, 0xFF);
34         }
35     }

```

34 shows a `Color` class that has internal instance fields named `redPart`, `bluePart`, and `greenPart`, and
35 static fields named `Red`, `Blue`, `Green`, and `White`

36 The use of static fields in this manner is not ideal. The fields are initialized at some point before they are
37 used, but after this initialization there is nothing to stop a client from changing them. Such a modification
38 could cause unpredictable errors in other programs that use `Color` and assume that the values do not
39 change. *Readonly fields* can be used to prevent such problems. Assignments to a readonly field can only
40 occur as part of the declaration, or in an instance constructor or static constructor in the same class. A static
41 readonly field can be assigned in a static constructor, and a non-static readonly field can be assigned in an
42 instance constructor. Thus, the `Color` class can be enhanced by adding the modifier `readonly` to the static
43 fields:

```

44     class Color
45     {
46         internal ushort redPart;
47         internal ushort bluePart;
48         internal ushort greenPart;
49
50         public Color(ushort red, ushort blue, ushort green) {
51             redPart = red;
52             bluePart = blue;
53             greenPart = green;
54         }
55     }

```



```

1         public static readonly Color Red = new Color(0xFF, 0, 0);
2         public static readonly Color Blue = new Color(0, 0xFF, 0);
3         public static readonly Color Green = new Color(0, 0, 0xFF);
4         public static readonly Color White = new Color(0xFF, 0xFF, 0xFF);
5     }

```

8.7.3 Methods

A *method* is a member that implements a computation or action that can be performed by an object or class. Methods have a (possibly empty) list of formal parameters, a return value (unless the method's *return-type* is `void`), and are either static or non-static. *Static methods* are accessed through the class. *Non-static methods*, which are also called *instance methods*, are accessed through instances of the class. The example

```

11     using System;
12     public class Stack
13     {
14         public static Stack Clone(Stack s) {...}
15         public static Stack Flip(Stack s) {...}
16         public object Pop() {...}
17         public void Push(object o) {...}
18         public override string ToString() {...}
19         ...
20     }
21     class Test
22     {
23         static void Main() {
24             Stack s = new Stack();
25             for (int i = 1; i < 10; i++)
26                 s.Push(i);
27             Stack flipped = Stack.Flip(s);
28             Stack cloned = Stack.Clone(s);
29             Console.WriteLine("Original stack: " + s.ToString());
30             Console.WriteLine("Flipped stack: " + flipped.ToString());
31             Console.WriteLine("Cloned stack: " + cloned.ToString());
32         }
33     }

```

shows a `Stack` that has several static methods (`Clone` and `Flip`) and several instance methods (`Pop`, `Push`, and `ToString`).

Methods can be overloaded, which means that multiple methods may have the same name so long as they have unique signatures. The signature of a method consists of the name of the method and the number, modifiers, and types of its formal parameters. The signature of a method does not include the return type. The example

```

40     using System;
41     class Test
42     {
43         static void F() {
44             Console.WriteLine("F()");
45         }
46         static void F(object o) {
47             Console.WriteLine("F(object)");
48         }
49         static void F(int value) {
50             Console.WriteLine("F(int)");
51         }
52         static void F(ref int value) {
53             Console.WriteLine("F(ref int)");
54         }

```

C# LANGUAGE SPECIFICATION

```
1     static void F(int a, int b) {
2         Console.WriteLine("F(int, int)");
3     }
4     static void F(int[] values) {
5         Console.WriteLine("F(int[])");
6     }
7     static void Main() {
8         F();
9         F(1);
10        int i = 10;
11        F(ref i);
12        F((object)1);
13        F(1, 2);
14        F(new int[] {1, 2, 3});
15    }
16 }
```

17 shows a class with a number of methods called F. The output produced is

```
18     F()
19     F(int)
20     F(ref int)
21     F(object)
22     F(int, int)
23     F(int[])
```

24 8.7.4 Properties

25 A *property* is a member that provides access to a characteristic of an object or a class. Examples of
26 properties include the length of a string, the size of a font, the caption of a window, the name of a customer,
27 and so on. Properties are a natural extension of fields. Both are named members with associated types, and
28 the syntax for accessing fields and properties is the same. However, unlike fields, properties do not denote
29 storage locations. Instead, properties have accessors that specify the statements to be executed when their
30 values are read or written.

31 Properties are defined with property declarations. The first part of a property declaration looks quite similar
32 to a field declaration. The second part includes a get accessor and/or a set accessor. In the example below,
33 the `Button` class defines a `Caption` property.

```
34     public class Button
35     {
36         private string caption;
37         public string Caption {
38             get {
39                 return caption;
40             }
41             set {
42                 caption = value;
43                 Repaint();
44             }
45         }
46     }
```

47 Properties that can be both read and written, such as `Caption`, include both get and set accessors. The get
48 accessor is called when the property's value is read; the set accessor is called when the property's value is
49 written. In a set accessor, the new value for the property is made available via an implicit parameter named
50 `value`.

51 The declaration of properties is relatively straightforward, but the real value of properties is seen when they
52 are used. For example, the `Caption` property can be read and written in the same way that fields can be read
53 and written:

```

1      Button b = new Button();
2      b.Caption = "ABC";           // set; causes repaint
3      string s = b.Caption;       // get
4      b.Caption += "DEF";         // get & set; causes repaint

```

8.7.5 Events

An *event* is a member that enables an object or class to provide notifications. A class defines an event by providing an event declaration (which resembles a field declaration, though with an added `event` keyword) and an optional set of event accessors. The type of this declaration must be a delegate type.

An instance of a delegate type encapsulates one or more callable entities. For instance methods, a callable entity consists of an instance and a method on that instance. For static methods, a callable entity consists of just a method. Given a delegate instance and an appropriate set of arguments, one can invoke all of that delegate instance's methods with that set of arguments.

In the example

```

14      public delegate void EventHandler(object sender, System.EventArgs e);
15      public class Button
16      {
17          public event EventHandler Click;
18          public void Reset() {
19              Click = null;
20          }
21      }

```

the `Button` class defines a `Click` event of type `EventHandler`. Inside the `Button` class, the `Click` member is exactly like a private field of type `EventHandler`. However, outside the `Button` class, the `Click` member can only be used on the left-hand side of the `+=` and `-=` operators. The `+=` operator adds a handler for the event, and the `-=` operator removes a handler for the event. The example

```

26      using System;
27      public class Form1
28      {
29          public Form1() {
30              // Add Button1_Click as an event handler for Button1's Click event
31              Button1.Click += new EventHandler(Button1_Click);
32          }
33          Button Button1 = new Button();
34          void Button1_Click(object sender, EventArgs e) {
35              Console.WriteLine("Button1 was clicked!");
36          }
37          public void Disconnect() {
38              Button1.Click -= new EventHandler(Button1_Click);
39          }
40      }

```

shows a `Form1` class that adds `Button1_Click` as an event handler for `Button1`'s `Click` event. In the `Disconnect` method, that event handler is removed.

For a simple event declaration such as

```

44      public event EventHandler Click;

```

the compiler automatically provides the implementation underlying the `+=` and `-=` operators.

An implementer who wants more control can get it by explicitly providing add and remove accessors. For example, the `Button` class could be rewritten as follows:

```

48      public class Button
49      {
50          private EventHandler handler;
51          public event EventHandler Click {
52              add { handler += value; }
53          }

```

```

1         remove { handler -= value; }
2     }
3 }

```

4 This change has no effect on client code, but allows the `Button` class more implementation flexibility. For
5 example, the event handler for `Click` need not be represented by a field.

6 8.7.6 Operators

7 An *operator* is a member that defines the meaning of an expression operator that can be applied to instances
8 of the class. There are three kinds of operators that can be defined: unary operators, binary operators, and
9 conversion operators.

10 The following example defines a `Digit` type that represents decimal digits—integral values between 0
11 and 9.

```

12     using System;
13     public struct Digit
14     {
15         byte value;
16         public Digit(byte value) {
17             if (value < 0 || value > 9) throw new ArgumentException();
18             this.value = value;
19         }
20         public Digit(int value): this((byte) value) {}
21         public static implicit operator byte(Digit d) {
22             return d.value;
23         }
24         public static explicit operator Digit(byte b) {
25             return new Digit(b);
26         }
27         public static Digit operator+(Digit a, Digit b) {
28             return new Digit(a.value + b.value);
29         }
30         public static Digit operator-(Digit a, Digit b) {
31             return new Digit(a.value - b.value);
32         }
33         public static bool operator==(Digit a, Digit b) {
34             return a.value == b.value;
35         }
36         public static bool operator!=(Digit a, Digit b) {
37             return a.value != b.value;
38         }
39         public override bool Equals(object value) {
40             if (value == null) return false;
41             if (GetType() == value.GetType()) return this == (Digit)value;
42             return false; }
43         public override int GetHashCode() {
44             return value.GetHashCode();
45         }
46         public override string ToString() {
47             return value.ToString();
48         }
49     }

```

```

1      class Test
2      {
3          static void Main() {
4              Digit a = (Digit) 5;
5              Digit b = (Digit) 3;
6              Digit plus = a + b;
7              Digit minus = a - b;
8              bool equals = (a == b);
9              Console.WriteLine("{0} + {1} = {2}", a, b, plus);
10             Console.WriteLine("{0} - {1} = {2}", a, b, minus);
11             Console.WriteLine("{0} == {1} = {2}", a, b, equals);
12         }
13     }

```

14 The `Digit` type defines the following operators:

- 15 • An implicit conversion operator from `Digit` to `byte`.
- 16 • An explicit conversion operator from `byte` to `Digit`.
- 17 • An addition operator that adds two `Digit` values and returns a `Digit` value.
- 18 • A subtraction operator that subtracts one `Digit` value from another, and returns a `Digit` value.
- 19 • The equality (`==`) and inequality (`!=`) operators, which compare two `Digit` values.

20 8.7.7 Indexers

21 An *indexer* is a member that enables an object to be indexed in the same way as an array. Whereas
 22 properties enable field-like access, indexers enable array-like access.

23 As an example, consider the `Stack` class presented earlier. The designer of this class might want to expose
 24 array-like access so that it is possible to inspect or alter the items on the stack without performing
 25 unnecessary `Push` and `Pop` operations. That is, class `Stack` is implemented as a linked list, but it also
 26 provides the convenience of array access.

27 Indexer declarations are similar to property declarations, with the main differences being that indexers are
 28 nameless (the “name” used in the declaration is `this`, since `this` is being indexed) and that indexers
 29 include indexing parameters. The indexing parameters are provided between square brackets. The example

```

30     using System;
31     public class Stack
32     {
33         private Node GetNode(int index) {
34             Node temp = first;
35             while (index > 0) {
36                 temp = temp.Next;
37                 index--;
38             }
39             return temp;
40         }
41         public object this[int index] {
42             get {
43                 if (!ValidIndex(index))
44                     throw new Exception("Index out of range.");
45                 else
46                     return GetNode(index).value;
47             }
48             set {
49                 if (!ValidIndex(index))
50                     throw new Exception("Index out of range.");
51                 else
52                     GetNode(index).value = value;
53             }
54         }
55         ...
56     }

```

```

1      class Test
2      {
3          static void Main() {
4              Stack s = new Stack();
5
6              s.Push(1);
7              s.Push(2);
8              s.Push(3);
9
10             s[0] = 33; // Changes the top item from 3 to 33
11             s[1] = 22; // Changes the middle item from 2 to 22
12             s[2] = 11; // Changes the bottom item from 1 to 11
13         }
14     }

```

13 shows an indexer for the `Stack` class.

14 8.7.8 Instance constructors

15 An *instance constructor* is a member that implements the actions required to initialize an instance of a class.

16 The example

```

17     using System;
18     class Point
19     {
20         public double x, y;
21         public Point() {
22             this.x = 0;
23             this.y = 0;
24         }
25         public Point(double x, double y) {
26             this.x = x;
27             this.y = y;
28         }
29         public static double Distance(Point a, Point b) {
30             double xdiff = a.x - b.x;
31             double ydiff = a.y - b.y;
32             return Math.Sqrt(xdiff * xdiff + ydiff * ydiff);
33         }
34         public override string ToString() {
35             return string.Format("{0}, {1}", x, y);
36         }
37     }
38     class Test
39     {
40         static void Main() {
41             Point a = new Point();
42             Point b = new Point(3, 4);
43             double d = Point.Distance(a, b);
44             Console.WriteLine("Distance from {0} to {1} is {2}", a, b, d);
45         }
46     }

```

47 shows a `Point` class that provides two public instance constructors, one of which takes no arguments, while
48 the other takes two `double` arguments.

49 If no instance constructor is supplied for a class, then an empty one with no parameters is automatically
50 provided.

51 8.7.9 Destructors

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

1 The example

```

2     using System;
3     class Point
4     {
5         public double x, y;
6         public Point(double x, double y) {
7             this.x = x;
8             this.y = y;
9         }
10        ~Point() {
11            Console.WriteLine("Destructed {0}", this);
12        }
13        public override string ToString() {
14            return string.Format("{0}, {1}", x, y);
15        }
16    }

```

17 shows a `Point` class with a destructor.

18 8.7.10 Static constructors

19 A *static constructor* is a member that implements the actions required to initialize a class. Static constructors cannot have parameters, they cannot have accessibility modifiers, and they cannot be called explicitly. The static constructor for a class is called automatically.

22 The example

```

23     using Personnel.Data;
24     class Employee
25     {
26         private static DataSet ds;
27         static Employee() {
28             ds = new DataSet(...);
29         }
30         public string Name;
31         public decimal Salary;
32         ...
33     }

```

34 shows an `Employee` class with a static constructor that initializes a static field.

35 8.7.11 Inheritance

36 Classes support single inheritance, and the type `object` is the ultimate base class for all classes.

37 The classes shown in earlier examples all implicitly derive from `object`. The example

```

38     using System;
39     class A
40     {
41         public void F() { Console.WriteLine("A.F"); }
42     }

```

43 shows a class `A` that implicitly derives from `object`. The example

```

44     class B: A
45     {
46         public void G() { Console.WriteLine("B.G"); }
47     }
48     class Test
49     {
50         static void Main() {
51             B b = new B();
52             b.F();           // Inherited from A
53             b.G();           // Introduced in B
54         }

```

```

1         A a = b;           // Treat a B as an A
2         a.F();
3     }
4 }

```

5 shows a class **B** that derives from **A**. The class **B** inherits **A**'s **F** method, and introduces a **G** method of its own.

6 Methods, properties, and indexers can be *virtual*, which means that their implementation can be overridden
7 in derived classes. The example

```

8     using System;
9     class A
10    {
11        public virtual void F() { Console.WriteLine("A.F"); }
12    }
13    class B: A
14    {
15        public override void F() {
16            base.F();
17            Console.WriteLine("B.F");
18        }
19    }
20    class Test
21    {
22        static void Main() {
23            B b = new B();
24            b.F();
25
26            A a = b;
27            a.F();
28        }
29    }

```

29 shows a class **A** with a virtual method **F**, and a class **B** that overrides **F**. The overriding method in **B** contains
30 a call, `base.F()`, which calls the overridden method in **A**.

31 A class can indicate that it is incomplete, and is intended only as a base class for other classes, by including
32 the modifier **abstract**. Such a class is called an *abstract class*. An abstract class can specify *abstract*
33 *members*—members that a non-abstract derived class must implement. The example

```

34    using System;
35    abstract class A
36    {
37        public abstract void F();
38    }
39    class B: A
40    {
41        public override void F() { Console.WriteLine("B.F"); }
42    }
43    class Test
44    {
45        static void Main() {
46            B b = new B();
47            b.F();
48
49            A a = b;
50            a.F();
51        }
52    }

```

52 introduces an abstract method **F** in the abstract class **A**. The non-abstract class **B** provides an implementation
53 for this method.

54 8.8 Structs

55 The list of similarities between classes and structs is long—structs can implement interfaces, and can have
56 the same kinds of members as classes. Structs differ from classes in several important ways, however:

1 structs are value types rather than reference types, and inheritance is not supported for structs. Struct values
 2 are stored “on the stack” or “in-line”. Careful programmers can sometimes enhance performance through
 3 judicious use of structs.

4 For example, the use of a struct rather than a class for a `Point` can make a large difference in the number of
 5 memory allocations performed at run time. The program below creates and initializes an array of 100 points.
 6 With `Point` implemented as a class, 101 separate objects are instantiated—one for the array and one each
 7 for the 100 elements.

```

8     class Point
9     {
10        public int x, y;
11        public Point(int x, int y) {
12            this.x = x;
13            this.y = y;
14        }
15    }
16    class Test
17    {
18        static void Main() {
19            Point[] points = new Point[100];
20            for (int i = 0; i < 100; i++)
21                points[i] = new Point(i, i*i);
22        }
23    }

```

24 If `Point` is instead implemented as a struct, as in

```

25    struct Point
26    {
27        public int x, y;
28        public Point(int x, int y) {
29            this.x = x;
30            this.y = y;
31        }
32    }

```

33 only one object is instantiated—the one for the array. The `Point` instances are allocated in-line within the
 34 array. This optimization can be misused. Using structs instead of classes can also make an application run
 35 slower or take up more memory, as passing a struct instance by value causes a copy of that struct to be
 36 created.

37 8.9 Interfaces

38 An interface defines a contract. A class or struct that implements an interface must adhere to its contract.
 39 Interfaces can contain methods, properties, events, and indexers as members.

40 The example

```

41    interface IExample
42    {
43        string this[int index] { get; set; }
44        event EventHandler E;
45        void F(int value);
46        string P { get; set; }
47    }
48    public delegate void EventHandler(object sender, EventArgs e);

```

49 shows an interface that contains an indexer, an event `E`, a method `F`, and a property `P`.

50 Interfaces may employ multiple inheritance. In the example

```

51    interface IControl
52    {
53        void Paint();
54    }

```

C# LANGUAGE SPECIFICATION

```
1     interface ITextBox: IControl
2     {
3         void SetText(string text);
4     }
5     interface IListBox: IControl
6     {
7         void SetItems(string[] items);
8     }
9     interface IComboBox: ITextBox, IListBox {}
```

10 the interface IComboBox inherits from both ITextBox and IListBox.

11 Classes and structs can implement multiple interfaces. In the example

```
12     interface IDataBound
13     {
14         void Bind(Binder b);
15     }
16     public class EditBox: Control, IControl, IDataBound
17     {
18         public void Paint() {...}
19         public void Bind(Binder b) {...}
20     }
```

21 the class EditBox derives from the class Control and implements both IControl and IDataBound.

22 In the previous example, the Paint method from the IControl interface and the Bind method from
23 IDataBound interface are implemented using public members on the EditBox class. C# provides an
24 alternative way of implementing these methods that allows the implementing class to avoid having these
25 members be public. Interface members can be implemented using a qualified name. For example, the
26 EditBox class could instead be implemented by providing IControl.Paint and IDataBound.Bind
27 methods.

```
28     public class EditBox: IControl, IDataBound
29     {
30         void IControl.Paint() {...}
31         void IDataBound.Bind(Binder b) {...}
32     }
```

33 Interface members implemented in this way are called *explicit interface members* because each member
34 explicitly designates the interface member being implemented. Explicit interface members can only be
35 called via the interface. For example, the EditBox's implementation of the Paint method can be called
36 only by casting to the IControl interface.

```
37     class Test
38     {
39         static void Main() {
40             EditBox editbox = new EditBox();
41             editbox.Paint(); // error: no such method
42             IControl control = editbox;
43             control.Paint(); // calls EditBox's Paint implementation
44         }
45     }
```

46 8.10 Delegates

47 Delegates enable scenarios that some other languages have addressed with function pointers. However,
48 unlike function pointers, delegates are object-oriented and type-safe.

49 A delegate declaration defines a class that is derived from the class System.Delegate. A delegate instance
50 encapsulates one or more methods, each of which is referred to as a *callable entity*. For instance methods, a
51 callable entity consists of an instance and a method on that instance. For static methods, a callable entity
52 consists of just a method. Given a delegate instance and an appropriate set of arguments, one can invoke all
53 of that delegate instance's methods with that set of arguments.

1 An interesting and useful property of a delegate instance is that it does not know or care about the classes of
 2 the methods it encapsulates; all that matters is that those methods be compatible (§22.1) with the delegate's
 3 type. This makes delegates perfectly suited for “anonymous” invocation. This is a powerful capability.

4 There are three steps in defining and using delegates: declaration, instantiation, and invocation. Delegates
 5 are declared using delegate declaration syntax. The example

```
6     delegate void SimpleDelegate();
```

7 declares a delegate named `SimpleDelegate` that takes no arguments and returns no result.

8 The example

```
9     class Test
10    {
11        static void F() {
12            System.Console.WriteLine("Test.F");
13        }
14        static void Main() {
15            SimpleDelegate d = new SimpleDelegate(F);
16            d();
17        }
18    }
```

19 creates a `SimpleDelegate` instance and then immediately calls it.

20 There is not much point in instantiating a delegate for a method and then immediately calling that method
 21 via the delegate, as it would be simpler to call the method directly. Delegates really show their usefulness
 22 when their anonymity is used. The example

```
23     void MultiCall(SimpleDelegate d, int count) {
24         for (int i = 0; i < count; i++)
25             d();
26     }
27 }
```

28 shows a `MultiCall` method that repeatedly calls a `SimpleDelegate`. The `MultiCall` method doesn't
 29 know or care about the type of the target method for the `SimpleDelegate`, what accessibility that method
 30 has, or whether or not that method is static. All that matters is that the target method is compatible (§22.1)
 31 with `SimpleDelegate`.

32 8.11 Enums

33 An enum type declaration defines a type name for a related group of symbolic constants. Enums are used for
 34 “multiple choice” scenarios, in which a runtime decision is made from a fixed number of choices that are
 35 known at compile-time.

36 The example

```
37     enum Color
38    {
39        Red,
40        Blue,
41        Green
42    }
43
44     class Shape
45    {
46        public void Fill(Color color) {
47            switch(color) {
48                case Color.Red:
49                    ...
50                    break;
51                case Color.Blue:
52                    ...
53                    break;
```

```

1         case Color.Green:
2             ...
3             break;
4         default:
5             break;
6     }
7 }
8

```

9 shows a `Color` enum and a method that uses this enum. The signature of the `Fill` method makes it clear
10 that the shape can be filled with one of the given colors.

11 The use of enums is superior to the use of integer constants—as is common in languages without enums—
12 because the use of enums makes the code more readable and self-documenting. The self-documenting nature
13 of the code also makes it possible for the development tool to assist with code writing and other “designer”
14 activities. For example, the use of `Color` rather than `int` for a parameter type enables smart code editors to
15 suggest `Color` values.

16 8.12 Namespaces and assemblies

17 The programs presented so far have stood on their own except for dependence on a few system-provided
18 classes such as `System.Console`. It is far more common, however, for real-world applications to consist of
19 several different pieces, each compiled separately. For example, a corporate application might depend on
20 several different components, including some developed internally and some purchased from independent
21 software vendors.

22 *Namespaces* and *assemblies* enable this component-based system. Namespaces provide a logical
23 organizational system. Namespaces are used both as an “internal” organization system for a program, and as
24 an “external” organization system—a way of presenting program elements that are exposed to other
25 programs.

26 *Assemblies* are used for physical packaging and deployment. An assembly may contain types, the executable
27 code used to implement these types, and references to other assemblies.

28 To demonstrate the use of namespaces and assemblies, this section revisits the “hello, world” program
29 presented earlier, and splits it into two pieces: a class library that provides messages and a console
30 application that displays them.

31 The class library will contain a single class named `HelloMessage`. The example

```

32     // HelloLibrary.cs
33     namespace CSharp.Introduction
34     {
35         public class HelloMessage
36         {
37             public string Message {
38                 get {
39                     return "hello, world";
40                 }
41             }
42         }
43     }

```

44 shows the `HelloMessage` class in a namespace named `CSharp.Introduction`. The `HelloMessage`
45 class provides a read-only property named `Message`. Namespaces can nest, and the declaration

```

46     namespace CSharp.Introduction
47     {...}

```

48 is shorthand for two levels of namespace nesting:

```

49     namespace CSharp
50     {
51         namespace Introduction
52         {...}
53     }

```

1 The next step in the componentization of “hello, world” is to write a console application that uses the
 2 `HelloMessage` class. The fully qualified name for the class—
 3 `CSharp.Introduction>HelloMessage`—could be used, but this name is quite long and unwieldy. An
 4 easier way is to use a *using namespace directive*, which makes it possible to use all of the types in a
 5 namespace without qualification. The example

```
6 // HelloApp.cs
7 using CSharp.Introduction;
8 class HelloApp
9 {
10     static void Main() {
11         HelloMessage m = new HelloMessage();
12         System.Console.WriteLine(m.Message);
13     }
14 }
```

15 shows a using namespace directive that refers to the `CSharp.Introduction` namespace. The occurrences
 16 of `HelloMessage` are shorthand for `CSharp.Introduction>HelloMessage`.

17 C# also enables the definition and use of aliases. A *using alias directive* defines an alias for a type. Such
 18 aliases can be useful in situation in which name collisions occur between two class libraries, or when a small
 19 number of types from a much larger namespace are being used. The example

```
20 using MessageSource = CSharp.Introduction>HelloMessage;
```

21 shows a using alias directive that defines `MessageSource` as an alias for the `HelloMessage` class.

22 The code we have written can be compiled into a class library containing the class `HelloMessage` and an
 23 application containing the class `HelloApp`. The details of this compilation step might differ based on the
 24 compiler or tool being used. A command-line compiler might enable compilation of a class library and an
 25 application that uses that library with the following command-line invocations:

```
26 csc /target:library HelloLibrary.cs
27 csc /reference:HelloLibrary.dll HelloApp.cs
```

28 which produce a class library named `HelloLibrary.dll` and an application named `HelloApp.exe`.

29 8.13 Versioning

30 *Versioning* is the process of evolving a component over time in a compatible manner. A new version of a
 31 component is *source compatible* with a previous version if code that depends on the previous version can,
 32 when recompiled, work with the new version. In contrast, a new version of a component is *binary*
 33 *compatible* if an application that depended on the old version can, without recompilation, work with the new
 34 version.

35 Most languages do not support binary compatibility at all, and many do little to facilitate source
 36 compatibility. In fact, some languages contain flaws that make it impossible, in general, to evolve a class
 37 over time without breaking at least some client code.

38 As an example, consider the situation of a base class author who ships a class named `Base`. In the first
 39 version, `Base` contains no method `F`. A component named `Derived` derives from `Base`, and introduces
 40 an `F`. This `Derived` class, along with the class `Base` on which it depends, is released to customers, who
 41 deploy to numerous clients and servers.

```
42 // Author A
43 namespace A
44 {
45     public class Base // version 1
46     {
47     }
48 }
```

C# LANGUAGE SPECIFICATION

```
1      // Author B
2      namespace B
3      {
4          class Derived: A.Base
5          {
6              public virtual void F() {
7                  System.Console.WriteLine("Derived.F");
8              }
9          }
10     }
```

11 So far, so good, but now the versioning trouble begins. The author of `Base` produces a new version, giving it
12 its own method `F`.

```
13     // Author A
14     namespace A
15     {
16         public class Base    // version 2
17         {
18             public virtual void F() { // added in version 2
19                 System.Console.WriteLine("Base.F");
20             }
21         }
22     }
```

23 This new version of `Base` should be both source and binary compatible with the initial version. (If it weren't
24 possible to simply add a method then a base class could never evolve.) Unfortunately, the new `F` in `Base`
25 makes the meaning of `Derived`'s `F` unclear. Did `Derived` mean to override `Base`'s `F`? This seems unlikely,
26 since when `Derived` was compiled, `Base` did not even have an `F`! Further, if `Derived`'s `F` does override
27 `Base`'s `F`, then it must adhere to the contract specified by `Base`—a contract that was unspecified when
28 `Derived` was written. In some cases, this is impossible. For example, `Base`'s `F` might require that overrides
29 of it always call the base. `Derived`'s `F` could not possibly adhere to such a contract.

30 C# addresses this versioning problem by requiring developers to state their intent clearly. In the original
31 code example, the code was clear, since `Base` did not even have an `F`. Clearly, `Derived`'s `F` is intended as a
32 new method rather than an override of a base method, since no base method named `F` exists.

33 If `Base` adds an `F` and ships a new version, then the intent of a binary version of `Derived` is still clear—
34 `Derived`'s `F` is semantically unrelated, and should not be treated as an override.

35 However, when `Derived` is recompiled, the meaning is unclear—the author of `Derived` may intend its `F` to
36 override `Base`'s `F`, or to hide it. Since the intent is unclear, the compiler produces a warning, and by default
37 makes `Derived`'s `F` hide `Base`'s `F`. This course of action duplicates the semantics for the case in which
38 `Derived` is not recompiled. The warning that is generated alerts `Derived`'s author to the presence of the
39 `F` method in `Base`.

40 If `Derived`'s `F` is semantically unrelated to `Base`'s `F`, then `Derived`'s author can express this intent—and,
41 in effect, turn off the warning—by using the new keyword in the declaration of `F`.

```
42     // Author A
43     namespace A
44     {
45         public class Base    // version 2
46         {
47             public virtual void F() { // added in version 2
48                 System.Console.WriteLine("Base.F");
49             }
50         }
51     }
```

```

1      // Author B
2      namespace B
3      {
4          class Derived: A.Base    // version 2a: new
5          {
6              new public virtual void F() {
7                  System.Console.WriteLine("Derived.F");
8              }
9          }
10     }

```

11 On the other hand, `Derived`'s author might investigate further, and decide that `Derived`'s `F` should
12 override `Base`'s `F`. This intent can be specified by using the `override` keyword, as shown below.

```

13     // Author A
14     namespace A
15     {
16         public class Base          // version 2
17         {
18             public virtual void F() { // added in version 2
19                 System.Console.WriteLine("Base.F");
20             }
21         }
22     }
23
24     // Author B
25     namespace B
26     {
27         class Derived: A.Base    // version 2b: override
28         {
29             public override void F() {
30                 base.F();
31                 System.Console.WriteLine("Derived.F");
32             }
33         }
34     }

```

34 The author of `Derived` has one other option, and that is to change the name of `F`, thus completely avoiding
35 the name collision. Although this change would break source and binary compatibility for `Derived`, the
36 importance of this compatibility varies depending on the scenario. If `Derived` is not exposed to other
37 programs, then changing the name of `F` is likely a good idea, as it would improve the readability of the
38 program—there would no longer be any confusion about the meaning of `F`.

39 8.14 Attributes

40 C# is an imperative language, but like all imperative languages it does have some declarative elements. For
41 example, the accessibility of a method in a class is specified by declaring it `public`, `protected`,
42 `internal`, `protected internal`, or `private`. C# generalizes this capability, so that programmers can
43 invent new kinds of declarative information, attach this declarative information to various program entities,
44 and retrieve this declarative information at run-time. Programs specify this additional declarative
45 information by defining and using attributes (§24).

46 For instance, a framework might define a `HelpAttribute` attribute that can be placed on program elements
47 such as classes and methods, enabling developers to provide a mapping from program elements to
48 documentation for them. The example

```

49     using System;
50     [AttributeUsage(AttributeTargets.All)]
51     public class HelpAttribute: Attribute
52     {
53         public HelpAttribute(string url) {
54             this.url = url;
55         }
56
57         public string Topic = null;
58         private string url;

```

C# LANGUAGE SPECIFICATION

```
1         public string Url {
2             get { return url; }
3         }
4     }
```

5 defines an attribute class named `HelpAttribute`, or `Help` for short, that has one positional parameter
6 (`string url`) and one named parameter (`string Topic`). Positional parameters are defined by the
7 formal parameters for public instance constructors of the attribute class, and named parameters are defined
8 by public non-static read-write fields and properties of the attribute class.

9 The example

```
10     [Help("http://www.mycompany.com/.../Class1.htm")]
11     public class Class1
12     {
13         [Help("http://www.mycompany.com/.../Class1.htm", Topic = "F")]
14         public void F() {}
15     }
```

16 shows several uses of the attribute `Help`.

17 Attribute information for a given program element can be retrieved at run-time by using reflection support.

18 The example

```
19     using System;
20     class Test
21     {
22         static void Main() {
23             Type type = typeof(Class1);
24             object[] arr = type.GetCustomAttributes(typeof(HelpAttribute),
25 true);
26             if (arr.Length == 0)
27                 Console.WriteLine("Class1 has no Help attribute.");
28             else {
29                 HelpAttribute ha = (HelpAttribute) arr[0];
30                 Console.WriteLine("Url = {0}, Topic = {1}", ha.Url, ha.Topic);
31             }
32         }
33     }
```

34 checks to see if `Class1` has a `Help` attribute, and writes out the associated `Topic` and `Url` values if the
35 attribute is present.

36 **End of informative text.**

37

9. Lexical structure

9.1 Programs

A C# *program* consists of one or more source files, known formally as *compilation units* (§16.1). 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.

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.

Conforming implementations must accept Unicode source files encoded with the UTF-8 encoding form (as defined by the Unicode standard), and transform them into a sequence of Unicode characters.

Implementations may choose to accept and transform additional character encoding schemes (such as UTF-16, UTF-32, or non-Unicode character mappings).

[*Note:* It is beyond the scope of this standard to define how a file using a character representation other than Unicode might be transformed into a sequence of Unicode characters. During such transformation, however, it is recommended that the usual line-separating character (or sequence) in the other character set be translated to the two-character sequence consisting of the Unicode carriage-return character followed by Unicode line-feed character. For the most part this transformation will have no visible effects; however, it will affect the interpretation of verbatim string literal tokens (§9.4.4.5). The purpose of this recommendation is to allow a verbatim string literal to produce the same character sequence when its source file is moved between systems that support differing non-Unicode character sets, in particular, those using differing character sequences for line-separation. *end note*]

9.2 Grammars

This specification presents the syntax of the C# programming language using two grammars. The *lexical grammar* (§9.2.1) defines how Unicode characters are combined to form line terminators, white space, comments, tokens, and pre-processing directives. The *syntactic grammar* (§9.2.2) defines how the tokens resulting from the lexical grammar are combined to form C# programs.

9.2.1 Lexical grammar

The lexical grammar of C# is presented in §9.3, §9.4, and §9.5. 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 (§9.4), white space (§9.3.3), comments (§9.3.2), and pre-processing directives (§9.5).

Every source file in a C# program must conform to the *input* production of the lexical grammar (§9.3).

9.2.2 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 (§16.1) of the syntactic grammar.

1 9.3 Lexical analysis

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

```
4     input::
5         input-sectionopt
6
7     input-section::
8         input-section-part
9         input-section input-section-part
10
11    input-section-part::
12        input-elementsopt new-line
13        pp-directive
14
15    input-elements::
16        input-element
17        input-elements input-element
18
19    input-element::
20        whitespace
21        comment
22        token
```

19 Five basic elements make up the lexical structure of a C# source file: Line terminators (§9.3.1), white space
20 (§9.3.3), comments (§9.3.2), tokens (§9.4), and pre-processing directives (§9.5). Of these basic elements,
21 only tokens are significant in the syntactic grammar of a C# program (§9.2.2).

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

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

30 9.3.1 Line terminators

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

```
32     new-line::
33         Carriage return character (U+000D)
34         Line feed character (U+000A)
35         Carriage return character (U+000D) followed by line feed character (U+000A)
36         Line separator character (U+2028)
37         Paragraph separator character (U+2029)
```

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

- 41 • If the last character of the source file is a Control-Z character (U+001A), this character is deleted.
- 42 • A carriage-return character (U+000D) is added to the end of the source file if that source file is non-
43 empty and if the last character of the source file is not a carriage return (U+000D), a line feed (U+000A),
44 a line separator (U+2028), or a paragraph separator (U+2029).

45 9.3.2 Comments

46 Two forms of comments are supported: delimited comments and single-line comments.

1 A **delimited comment** begins with the characters `/*` and ends with the characters `*/`. Delimited comments
 2 can occupy a portion of a line, a single line, or multiple lines. [*Example:* The example

```

3     /* Hello, world program
4         This program writes "hello, world" to the console
5     */
6     class Hello
7     {
8         static void Main() {
9             System.Console.WriteLine("hello, world");
10        }
11    }

```

12 includes a delimited comment. *end example*]

13 A **single-line comment** begins with the characters `//` and extends to the end of the line. [*Example:* The
 14 example

```

15     // Hello, world program
16     // This program writes "hello, world" to the console
17     //
18     class Hello // any name will do for this class
19     {
20         static void Main() { // this method must be named "Main"
21             System.Console.WriteLine("hello, world");
22         }
23     }

```

24 shows several single-line comments. *end example*]

25 *comment::*

26 *single-line-comment*

27 *delimited-comment*

28 *single-line-comment::*

29 *// input-characters_{opt}*

30 *input-characters::*

31 *input-character*

32 *input-characters input-character*

33 *input-character::*

34 Any Unicode character except a *new-line-character*

35 *new-line-character::*

36 Carriage return character (U+000D)

37 Line feed character (U+000A)

38 Line separator character (U+2028)

39 Paragraph separator character (U+2029)

40 *delimited-comment::*

41 */* delimited-comment-characters_{opt} */*

42 *delimited-comment-characters::*

43 *delimited-comment-character*

44 *delimited-comment-characters delimited-comment-character*

45 *delimited-comment-character::*

46 *not-asterisk*

47 ** not-slash*

48 *not-asterisk::*

49 Any Unicode character except ***

50 *not-slash::*

51 Any Unicode character except */*

- 1 Comments do not nest. The character sequences `/*` and `*/` have no special meaning within a single-line
 2 comment, and the character sequences `//` and `/*` have no special meaning within a delimited comment.
 3 Comments are not processed within character and string literals.

4 9.3.3 White space

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

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

12 9.4 Tokens

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

```
15     token::
16         identifier
17         keyword
18         integer-literal
19         real-literal
20         character-literal
21         string-literal
22         operator-or-punctuator
```

23 9.4.1 Unicode escape sequences

24 A Unicode escape sequence represents a Unicode character. Unicode escape sequences are processed in
 25 identifiers (§9.4.2), regular string literals (§9.4.4.5), and character literals (§9.4.4.4). A Unicode character
 26 escape is not processed in any other location (for example, to form an operator, punctuator, or keyword).

```
27     unicode-escape-sequence::
28         \u hex-digit hex-digit hex-digit hex-digit
29         \U hex-digit hex-digit hex-digit hex-digit hex-digit hex-digit hex-digit hex-digit
```

30 A Unicode escape sequence represents the single Unicode character formed by the hexadecimal number
 31 following the “\u” or “\U” characters. Since C# uses a 16-bit encoding of Unicode characters in characters
 32 and string values, a Unicode character in the range U+10000 to U+10FFFF is represented using two Unicode
 33 surrogate characters. Unicode characters with code points above 0x10FFFF are not supported.

34 Multiple translations are not performed. For instance, the string literal “\u005Cu005C” is equivalent to
 35 “\u005C” rather than “\”. [Note: The Unicode value \u005C is the character “\”. end note]

36 [Example: The example

```
37     class Class1
38     {
39         static void Test(bool \u0066) {
40             char c = '\u0066';
41             if (\u0066)
42                 System.Console.WriteLine(c.ToString());
43         }
44     }
```

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

```

1      class Class1
2      {
3          static void Test(bool f) {
4              char c = 'f';
5              if (f)
6                  System.Console.WriteLine(c.ToString());
7          }
8      }

```

9 *end example]*

10 9.4.2 Identifiers

11 The rules for identifiers rules given in this section correspond exactly to those recommended by the Unicode
12 Standard Annex 15 except that underscore is allowed as an initial character (as is traditional in the
13 C programming language), Unicode escape sequences are permitted in identifiers, and the “@” character is
14 allowed as a prefix to enable keywords to be used as identifiers.

15 *identifier::*

16 *available-identifier*
17 *@ identifier-or-keyword*

18 *available-identifier::*

19 *An identifier-or-keyword that is not a keyword*

20 *identifier-or-keyword::*

21 *identifier-start-character identifier-part-characters_{opt}*

22 *identifier-start-character::*

23 *letter-character*
24 *_ (the underscore character U+005F)*

25 *identifier-part-characters::*

26 *identifier-part-character*
27 *identifier-part-characters identifier-part-character*

28 *identifier-part-character::*

29 *letter-character*
30 *decimal-digit-character*
31 *connecting-character*
32 *combining-character*
33 *formatting-character*

34 *letter-character::*

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

37 *combining-character::*

38 *A Unicode character of classes Mn or Mc*
39 *A unicode-escape-sequence representing a character of classes Mn or Mc*

40 *decimal-digit-character::*

41 *A Unicode character of the class Nd*
42 *A unicode-escape-sequence representing a character of the class Nd*

43 *connecting-character::*

44 *A Unicode character of the class Pc*
45 *A unicode-escape-sequence representing a character of the class Pc*

46 *formatting-character::*

47 *A Unicode character of the class Cf*
48 *A unicode-escape-sequence representing a character of the class Cf*

C# LANGUAGE SPECIFICATION

1 [Note: For information on the Unicode character classes mentioned above, see *The Unicode Standard*,
2 *Version 3.0*, §4.5.) *end note*]

3 [Example: Examples of valid identifiers include “identifier1”, “_identifier2”, and “@if”. *end*
4 *example*]

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

8 The prefix “@” enables the use of keywords as identifiers, which is useful when interfacing with other
9 programming languages. The character @ is not actually part of the identifier, so the identifier might be seen
10 in other languages as a normal identifier, without the prefix. An identifier with an @ prefix is called a
11 **verbatim identifier**. [Note: Use of the @ prefix for identifiers that are not keywords is permitted, but strongly
12 discouraged as a matter of style. *end note*]

13 [Example: The example:

```
14     class @class
15     {
16         public static void @static(bool @bool) {
17             if (@bool)
18                 System.Console.WriteLine("true");
19             else
20                 System.Console.WriteLine("false");
21         }
22     }
23     class Class1
24     {
25         static void M() {
26             cl\u0061ss.st\u0061tic(true);
27         }
28     }
```

29 defines a class named “class” with a static method named “static” that takes a parameter named
30 “bool”. Note that since Unicode escapes are not permitted in keywords, the token “cl\u0061ss” is an
31 identifier, and is the same identifier as “@class”. *end example*]

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

- 34 • The prefix “@”, if used, is removed.
- 35 • Each *unicode-escape-sequence* is transformed into its corresponding Unicode character.
- 36 • Any *formatting-characters* are removed.

37 Identifiers containing two consecutive underscore characters (U+005F) are reserved for use by the
38 implementation; however, no diagnostic is required if such an identifier is defined. [Note: For example, an
39 implementation might provide extended keywords that begin with two underscores. *end note*]

40 9.4.3 Keywords

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

```

1      keyword:: one of
2      abstract    as      base      bool      break
3      byte        case    catch    char      checked
4      class       const  continue decimal  default
5      delegate    do      double   else      enum
6      event       explicit extern  false    finally
7      fixed       float   for      foreach   goto
8      if          implicit in      int      interface
9      internal    is      lock     long     namespace
10     new         null    object   operator  out
11     override    params private protected public
12     readonly    ref     return   sbyte    sealed
13     short       sizeof  stackalloc static    string
14     struct      switch  this     throw    true
15     try         typeof  uint     ulong    unchecked
16     unsafe      ushort  using    virtual  void
17     volatile    while

```

18 In some places in the grammar, specific identifiers have special meaning, but are not keywords. [Note: For
19 example, within a property declaration, the “get” and “set” identifiers have special meaning (§17.6.2). An
20 identifier other than `get` or `set` is never permitted in these locations, so this use does not conflict with a use
21 of these words as identifiers. *end note*]

22 9.4.4 Literals

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

```

24     literal::
25         boolean-literal
26         integer-literal
27         real-literal
28         character-literal
29         string-literal
30         null-literal

```

31 9.4.4.1 Boolean literals

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

```

33     boolean-literal::
34         true
35         false

```

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

37 9.4.4.2 Integer literals

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

```

40     integer-literal::
41         decimal-integer-literal
42         hexadecimal-integer-literal
43
44     decimal-integer-literal::
45         decimal-digits integer-type-suffixopt
46
47     decimal-digits::
48         decimal-digit
49         decimal-digits decimal-digit

```

C# LANGUAGE SPECIFICATION

1 *decimal-digit*:: one of
2 0 1 2 3 4 5 6 7 8 9
3 *integer-type-suffix*:: one of
4 U u L l UL Ul uL ul LU Lu lU lu
5 *hexadecimal-integer-literal*::
6 0x *hex-digits integer-type-suffix_{opt}*
7 0X *hex-digits integer-type-suffix_{opt}*
8 *hex-digits*::
9 *hex-digit*
10 *hex-digits hex-digit*
11 *hex-digit*:: one of
12 0 1 2 3 4 5 6 7 8 9 A B C D E F a b c d e f

13 The type of an integer literal is determined as follows:

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

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

23 [*Note*: As a matter of style, it is suggested that “L” be used instead of “l” when writing literals of type `long`,
24 since it is easy to confuse the letter “l” with the digit “1”. *end note*]

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

- 27 • When a *decimal-integer-literal* with the value 2147483648 (2^{31}) and no *integer-type-suffix* appears as
28 the token immediately following a unary minus operator token (§14.6.2), the result is a constant of type
29 `int` with the value -2147483648 (-2^{31}). In all other situations, such a *decimal-integer-literal* is of type
30 `uint`.
- 31 • When a *decimal-integer-literal* with the value 9223372036854775808 (2^{63}) and no *integer-type-suffix* or
32 the *integer-type-suffix* L or l appears as the token immediately following a unary minus operator token
33 (§14.6.2), the result is a constant of type `long` with the value -9223372036854775808 (-2^{63}). In all
34 other situations, such a *decimal-integer-literal* is of type `ulong`.

35 9.4.4.3 Real literals

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

37 *real-literal*::
38 *decimal-digits* . *decimal-digits exponent-part_{opt} real-type-suffix_{opt}*
39 . *decimal-digits exponent-part_{opt} real-type-suffix_{opt}*
40 *decimal-digits exponent-part real-type-suffix_{opt}*
41 *decimal-digits real-type-suffix*
42 *exponent-part*::
43 e *sign_{opt} decimal-digits*
44 E *sign_{opt} decimal-digits*


```

1      sign:: one of
2          + -
3
4      real-type-suffix:: one of
5          F f D d M m

```

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

- 7 • A real literal suffixed by F or f is of type `float`. [*Example*: For example, the literals `1f`, `1.5f`, `1e10f`,
8 and `123.456F` are all of type `float`. *end example*]
- 9 • A real literal suffixed by D or d is of type `double`. [*Example*: For example, the literals `1d`, `1.5d`,
10 `1e10d`, and `123.456D` are all of type `double`. *end example*]
- 11 • A real literal suffixed by M or m is of type `decimal`. [*Example*: For example, the literals `1m`, `1.5m`,
12 `1e10m`, and `123.456M` are all of type `decimal`. *end example*] This literal is converted to a `decimal`
13 value by taking the exact value, and, if necessary, rounding to the nearest representable value using
14 banker's rounding (§11.1.6). Any scale apparent in the literal is preserved unless the value is rounded or
15 the value is zero (in which latter case the sign and scale will be 0). [*Note*: Hence, the literal `2.900m` will
16 be parsed to form the `decimal` with sign 0, coefficient 2900, and scale 3. *end note*]

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

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

9.4.4.4 Character literals

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

```

22      character-literal::
23          ' character '
24
25      character::
26          single-character
27          simple-escape-sequence
28          hexadecimal-escape-sequence
29          unicode-escape-sequence
30
31      single-character::
32          Any character except ' (U+0027), \ (U+005C), and new-line-character
33
34      simple-escape-sequence:: one of
35          \' \" \\ \0 \a \b \f \n \r \t \v
36
37      hexadecimal-escape-sequence::
38          \x hex-digit hex-digitopt hex-digitopt hex-digitopt

```

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

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

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

A Unicode character escape sequence (§9.4.1) in a character literal must be in the range U+0000 to U+FFFF.

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

42

Escape sequence	Character name	Unicode encoding
\'	Single quote	0x0027
\"	Double quote	0x0022
\\	Backslash	0x005C
\0	Null	0x0000
\a	Alert	0x0007
\b	Backspace	0x0008
\f	Form feed	0x000C
\n	New line	0x000A
\r	Carriage return	0x000D
\t	Horizontal tab	0x0009
\v	Vertical tab	0x000B

1
2 The type of a *character-literal* is `char`.

3 9.4.4.5 String literals

4 C# supports two forms of string literals: *regular string literals* and *verbatim string literals*. A regular string
5 literal consists of zero or more characters enclosed in double quotes, as in "hello, world", and may
6 include both simple escape sequences (such as \t for the tab character), and hexadecimal and Unicode
7 escape sequences.

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

13 *string-literal*::

14 *regular-string-literal*
15 *verbatim-string-literal*

16 *regular-string-literal*::

17 " *regular-string-literal-characters*_{opt} "

18 *regular-string-literal-characters*::

19 *regular-string-literal-character*
20 *regular-string-literal-characters* *regular-string-literal-character*

21 *regular-string-literal-character*::

22 *single-regular-string-literal-character*
23 *simple-escape-sequence*
24 *hexadecimal-escape-sequence*
25 *unicode-escape-sequence*

26 *single-regular-string-literal-character*::

27 Any character except " (U+0022), \ (U+005C), and *new-line-character*

28 *verbatim-string-literal*::

29 @" *verbatim -string-literal-characters*_{opt} "

```

1      verbatim-string-literal-characters::
2          verbatim-string-literal-character
3          verbatim-string-literal-characters verbatim-string-literal-character
4
5      verbatim-string-literal-character::
6          single-verbatim-string-literal-character
7          quote-escape-sequence
8
9      single-verbatim-string-literal-character::
10         any character except "
11
12     quote-escape-sequence::
13         ""

```

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

14 [Example: The example

```

15         string a = "Happy birthday, Joel";           // Happy birthday, Joel
16         string b = @"Happy birthday, Joel";         // Happy birthday, Joel
17         string c = "hello \t world";                 // hello      world
18         string d = @"hello \t world";                 // hello \t world
19         string e = "Joe said \"Hello\" to me";       // Joe said "Hello" to me
20         string f = @"Joe said ""Hello"" to me";      // Joe said "Hello" to me
21         string g = "\\server\share\file.txt";        // \\server\share\file.txt
22         string h = @"\\server\share\file.txt";      // \\server\share\file.txt
23         string i = "one\r\ntwo\r\nthree";
24         string j = @"one
25         two
26         three";

```

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

30 [Note: Since a hexadecimal escape sequence can have a variable number of hex digits, the string literal
31 "\x123" contains a single character with hex value 123. To create a string containing the character with hex
32 value 12 followed by the character 3, one could write "\x00123" or "\x12" + "3" instead. *end note*]

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

34 Each string literal does not necessarily result in a new string instance. When two or more string literals that
35 are equivalent according to the string equality operator (§14.9.7), appear in the same assembly, these string
36 literals refer to the same string instance. [Example: For instance, the output produced by

```

37         class Test
38         {
39             static void Main() {
40                 object a = "hello";
41                 object b = "hello";
42                 System.Console.WriteLine(a == b);
43             }
44         }

```

45 is `True` because the two literals refer to the same string instance. *end example*]

46 9.4.4.6 The null literal

```

47         null-literal::
48             null

```

49 The type of a *null-literal* is the null type.

1 9.4.5 Operators and punctuators

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

5 *operator-or-punctuator::* one of
 6 { } [] () . , : ;
 7 + - * / % & | ^ ! ~
 8 = < > ? ++ -- && || << >>
 9 == != <= >= += -= *= /= %= &=
 10 |= ^= <<= >>= ->

11 9.5 Pre-processing directives

12 The pre-processing directives provide the ability to conditionally skip sections of source files, to report error
 13 and warning conditions, and to delineate distinct regions of source code. [*Note:* The term “pre-processing
 14 directives” is used only for consistency with the C and C++ programming languages. In C#, there is no
 15 separate pre-processing step; pre-processing directives are processed as part of the lexical analysis phase.
 16 *end note*]

17 *pp-directive::*
 18 *pp-declaration*
 19 *pp-conditional*
 20 *pp-line*
 21 *pp-diagnostic*
 22 *pp-region*

23 The following pre-processing directives are available:

- 24 • `#define` and `#undef`, which are used to define and undefine, respectively, conditional compilation
 25 symbols (§9.5.3).
- 26 • `#if`, `#elif`, `#else`, and `#endif`, which are used to conditionally skip sections of source code (§9.5.1).
- 27 • `#line`, which is used to control line numbers emitted for errors and warnings (§9.5.7).
- 28 • `#error` and `#warning`, which are used to issue errors and warnings, respectively (§9.5.5).
- 29 • `#region` and `#endregion`, which are used to explicitly mark sections of source code (§9.5.6).

30 A pre-processing directive always occupies a separate line of source code and always begins with a
 31 `#` character and a pre-processing directive name. White space may occur before the `#` character and between
 32 the `#` character and the directive name.

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

36 Pre-processing directives are not tokens and are not part of the syntactic grammar of C#. However, pre-
 37 processing directives can be used to include or exclude sequences of tokens and can in that way affect the
 38 meaning of a C# program. For example, when compiled, the program

```
39     #define A
40     #undef B
41
42     class C
43     {
44     #if A
45         void F() {}
46     #else
47         void G() {}
48     #endif
```

```

1      #if B
2          void H() {}
3      #else
4          void I() {}
5      #endif
6      }

```

7 results in the exact same sequence of tokens as the program

```

8      class C
9      {
10         void F() {}
11         void I() {}
12     }

```

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

14 9.5.1 Conditional compilation symbols

15 The conditional compilation functionality provided by the `#if`, `#elif`, `#else`, and `#endif` directives is
 16 controlled through pre-processing expressions (§9.5.2) and conditional compilation symbols.

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

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

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

29 9.5.2 Pre-processing expressions

30 Pre-processing expressions can occur in `#if` and `#elif` directives. The operators `!`, `==`, `!=`, `&&` and `||` are
 31 permitted in pre-processing expressions, and parentheses may be used for grouping.

32 *pp-expression::*
 33 *whitespace_{opt} pp-or-expression whitespace_{opt}*

34 *pp-or-expression::*
 35 *pp-and-expression*
 36 *pp-or-expression whitespace_{opt} || whitespace_{opt} pp-and-expression*

37 *pp-and-expression::*
 38 *pp-equality-expression*
 39 *pp-and-expression whitespace_{opt} && whitespace_{opt} pp-equality-expression*

40 *pp-equality-expression::*
 41 *pp-unary-expression*
 42 *pp-equality-expression whitespace_{opt} == whitespace_{opt} pp-unary-expression*
 43 *pp-equality-expression whitespace_{opt} != whitespace_{opt} pp-unary-expression*

44 *pp-unary-expression::*
 45 *pp-primary-expression*
 46 *! whitespace_{opt} pp-unary-expression*

```

1      pp-primary-expression::
2          true
3          false
4          conditional-symbol
5          ( whitespaceopt pp-expression whitespaceopt )

```

6 When referenced in a pre-processing expression, a defined conditional compilation symbol has the boolean
7 value `true`, and an undefined conditional compilation symbol has the boolean value `false`.

8 Evaluation of a pre-processing expression always yields a boolean value. The rules of evaluation for a pre-
9 processing expression are the same as those for a constant expression (§14.15), except that the only user-
10 defined entities that can be referenced are conditional compilation symbols.

11 9.5.3 Declaration directives

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

```

13      pp-declaration::
14          whitespaceopt # whitespaceopt define whitespace conditional-symbol pp-new-line
15          whitespaceopt # whitespaceopt undef whitespace conditional-symbol pp-new-line
16
17      pp-new-line::
18          whitespaceopt single-line-commentopt new-line

```

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

22 Any `#define` and `#undef` directives in a source file must occur before the first *token* (§9.4) in the source
23 file; otherwise a compile-time error occurs. In intuitive terms, `#define` and `#undef` directives must
24 precede any “real code” in the source file.

25 [*Example*: The example:

```

26      #define Enterprise
27      #if Professional || Enterprise
28          #define Advanced
29      #endif
30      namespace Megacorp.Data
31      {
32          #if Advanced
33          class PivotTable {...}
34          #endif
35      }

```

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

37 *end example*]

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

```

39      #define A
40      namespace N
41      {
42          #define B
43          #if B
44          class Class1 {}
45          #endif
46      }

```

47 *end example*]

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

```

1      #define A
2      #define A

```

3 For compilers that allow conditional compilation symbols to be defined as compilation options, an
4 alternative way for such redefinition to occur is to define the symbol as a compiler option as well as in the
5 source. *end example*]

6 A `#undef` may “undefine” a conditional compilation symbol that is not defined. [*Example*: The example
7 below defines a conditional compilation symbol A and then undefines it twice; although the second `#undef`
8 has no effect, it is still valid.

```

9      #define A
10     #undef A
11     #undef A

```

12 *end example*]

13 9.5.4 Conditional compilation directives

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

```

15     pp-conditional::
16         pp-if-section pp-elif-sectionsopt pp-else-sectionopt pp-endif
17
18     pp-if-section::
19         whitespaceopt # whitespaceopt if whitespace pp-expression pp-new-line conditional-
20         sectionopt
21
22     pp-elif-sections::
23         pp-elif-section
24         pp-elif-sections pp-elif-section
25
26     pp-elif-section::
27         whitespaceopt # whitespaceopt elif whitespace pp-expression pp-new-line conditional-
28         sectionopt
29
30     pp-else-section::
31         whitespaceopt # whitespaceopt else pp-new-line conditional-sectionopt
32
33     pp-endif::
34         whitespaceopt # whitespaceopt endif pp-new-line
35
36     conditional-section::
37         input-section
38         skipped-section
39
40     skipped-section::
41         skipped-section-part
42         skipped-section skipped-section-part
43
44     skipped-section-part::
45         skipped-charactersopt new-line
46         pp-directive
47
48     skipped-characters::
49         whitespaceopt not-number-sign input-charactersopt
50
51     not-number-sign::
52         Any input-character except #

```

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

C# LANGUAGE SPECIFICATION

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

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

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

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

16 [Example: The following example illustrates how conditional compilation directives can nest:

```
17     #define Debug    // Debugging on
18     #undef Trace    // Tracing off
19
20     class PurchaseTransaction
21     {
22         void Commit() {
23             #if Debug
24                 CheckConsistency();
25                 #if Trace
26                     writeToLog(this.ToString());
27                 #endif
28             #endif
29             CommitHelper();
30         }
31     }
```

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

```
33     #define Debug    // Debugging on
34
35     class PurchaseTransaction
36     {
37         void Commit() {
38             #if Debug
39                 CheckConsistency();
40             #else
41                 /* Do something else
42             #endif
43         }
44     }
```

44 Note, however, that pre-processing directives are required to be lexically correct even in skipped sections of
45 source code.

46 Pre-processing directives are not processed when they appear inside multi-line input elements. For example,
47 the program:


```

1      class Hello
2      {
3          static void Main() {
4              System.Console.WriteLine(@"hello,
5              #if Debug
6                  world
7              #else
8                  Nebraska
9              #endif
10             ");
11         }
12     }

```

13 results in the output:

```

14     hello,
15     #if Debug
16         world
17     #else
18         Nebraska
19     #endif

```

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

```

22     #if X
23         /*
24         #else
25         /* */ class Q { }
26     #endif

```

27 always produces the same token stream (`class Q { }`), regardless of whether or not `X` is defined. If `X` is defined, the only processed directives are `#if` and `#endif`, due to the multi-line comment. If `X` is undefined, then three directives (`#if`, `#else`, `#endif`) are part of the directive set. *end example*

30 9.5.5 Diagnostic directives

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

```

33     pp-diagnostic::
34         whitespaceopt # whitespaceopt error pp-message
35         whitespaceopt # whitespaceopt warning pp-message
36
37     pp-message::
38         new-line
39         whitespace input-charactersopt new-line

```

39 [Example: The example

```

40     #warning Code review needed before check-in
41     #if Debug && Retail
42         #error A build can't be both debug and retail
43     #endif
44     class Test {...}

```

45 always produces a warning (“Code review needed before check-in”), and produces a compile-time error if the pre-processing identifiers `Debug` and `Retail` are both defined. Note that a *pp-message* can contain arbitrary text; specifically, it need not contain well-formed tokens, as shown by the single quote in the word `can't`. *end example*

49 9.5.6 Region control

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

```

51     pp-region::
52         pp-start-region conditional-sectionopt pp-end-region

```

C# LANGUAGE SPECIFICATION

```
1      pp-start-region::
2          whitespaceopt # whitespaceopt region pp-message
3
4      pp-end-region::
5          whitespaceopt # whitespaceopt endregion pp-message
```

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

9 The lexical processing of a region:

```
10      #region
11      ...
12      #endregion
```

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

```
14      #if true
15      ...
16      #endif
```

17 9.5.7 Line directives

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

20 [Note: Line directives are most commonly used in meta-programming tools that generate C# source code
21 from some other text input. *end note*]

```
22      pp-line::
23          whitespaceopt # whitespaceopt line whitespace line-indicator pp-new-line
24
25      line-indicator::
26          decimal-digits whitespace file-name
27          decimal-digits
28          default
29
30      file-name::
31          " file-name-characters "
32
33      file-name-characters::
34          file-name-character
35          file-name-characters file-name-character
36
37      file-name-character::
38          Any character except " (U+0022), and new-line
```

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

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

40 [Note: Note that a *file-name* differs from a regular string literal in that escape characters are not processed;
41 the ‘\’ character simply designates an ordinary back-slash character within a *file-name*. *end note*]

10. Basic concepts

1

2 10.1 Application startup

3 *Application startup* occurs when the execution environment calls a designated method, which is referred to
4 as the application's *entry point*. This entry point method is always named `Main`, and shall have one of the
5 following signatures:

```
6     static void Main() {...}
7     static void Main(string[] args) {...}
8     static int Main() {...}
9     static int Main(string[] args) {...}
```

10 As shown, the entry point may optionally return an `int` value. This return value is used in application
11 termination (§10.2).

12 The entry point may optionally have one formal parameter, and this formal parameter may have any name. If
13 such a parameter is declared, it must obey the following constraints:

- 14 • The implementation shall ensure that the value of this parameter is not `null`.
- 15 • Let `args` be the name of the parameter. If the length of the array designated by `args` is greater than
16 zero, the array members `args[0]` through `args[args.Length-1]`, inclusive, must refer to strings,
17 called *application parameters*, which are given implementation-defined values by the host environment
18 prior to application startup. The intent is to supply to the application information determined prior to
19 application startup from elsewhere in the hosted environment. If the host environment is not capable of
20 supplying strings with letters in both uppercase and lowercase, the implementation shall ensure that the
21 strings are received in lowercase. [*Note: On systems supporting a command line, application parameters*
22 *correspond to what are generally known as command-line arguments. end note*]

23 Since C# supports method overloading, a class or struct may contain multiple definitions of some method,
24 provided each has a different signature. However, within a single program, no class or struct shall contain
25 more than one method called `Main` whose definition qualifies it to be used as an application entry point.
26 Other overloaded versions of `Main` are permitted, however, provided they have more than one parameter, or
27 their only parameter is other than type `string[]`.

28 An application can be made up of multiple classes or structs. It is possible for more than one of these classes
29 or structs to contain a method called `Main` whose definition qualifies it to be used as an application entry
30 point. In such cases, one of these `Main` methods must be chosen as the entry point so that application startup
31 can occur. This choice of an entry point is beyond the scope of this specification—no mechanism for
32 specifying or determining an entry point is provided.

33 In C#, every method must be defined as a member of a class or struct. Ordinarily, the declared accessibility
34 (§10.5.1) of a method is determined by the access modifiers (§17.2.3) specified in its declaration, and
35 similarly the declared accessibility of a type is determined by the access modifiers specified in its
36 declaration. In order for a given method of a given type to be callable, both the type and the member must be
37 accessible. However, the application entry point is a special case. Specifically, the execution environment
38 can access the application's entry point regardless of its declared accessibility and regardless of the declared
39 accessibility of its enclosing type declarations.

40 In all other respects, entry point methods behave like those that are not entry points.

41 10.2 Application termination

42 *Application termination* returns control to the execution environment.

1 If the return type of the application's entry point method is `int`, the value returned serves as the
 2 application's *termination status code*. The purpose of this code is to allow communication of success or
 3 failure to the execution environment.

4 If the return type of the entry point method is `void`, reaching the right brace (`}`) which terminates that
 5 method, or executing a `return` statement that has no expression, results in a termination status code of 0.

6 Prior to an application's termination, destructors for all of its objects that have not yet been garbage
 7 collected are called, unless such cleanup has been suppressed (by a call to the library method
 8 `GC.SuppressFinalize`, for example).

9 **10.3 Declarations**

10 Declarations in a C# program define the constituent elements of the program. C# programs are organized
 11 using namespaces (§16), which can contain type declarations and nested namespace declarations. Type
 12 declarations (§16.5) are used to define classes (§17), structs (§18), interfaces (§20), enums (§21), and
 13 delegates (§22). The kinds of members permitted in a type declaration depend on the form of the type
 14 declaration. For instance, class declarations can contain declarations for constants (§17.3), fields (§17.4),
 15 methods (§17.5), properties (§17.6), events (§17.7), indexers (§17.8), operators (§17.9), instance
 16 constructors (§17.10), destructors (§17.12), static constructors (§17.11), and nested types.

17 A declaration defines a name in the *declaration space* to which the declaration belongs. Except for
 18 overloaded members (§10.6), it is a compile-time error to have two or more declarations that introduce
 19 members with the same name in a declaration space. However, no diagnostic is required if the declaration
 20 space is a namespace for the global declaration space and the conflicting declarations are in separate
 21 programs. It is never possible for a declaration space to contain different kinds of members with the same
 22 name. For example, a declaration space can never contain a field and a method by the same name.

23 There are several different types of declaration spaces, as described in the following.

- 24 • Within all source files of a program, *namespace-member-declarations* with no enclosing *namespace-*
 25 *declaration* are members of a single combined declaration space called the *global declaration space*.
- 26 • Within all source files of a program, *namespace-member-declarations* within *namespace-declarations*
 27 that have the same fully qualified namespace name are members of a single combined declaration space.
- 28 • Each class, struct, or interface declaration creates a new declaration space. Names are introduced into
 29 this declaration space through *class-member-declarations*, *struct-member-declarations*, or *interface-*
 30 *member-declarations*. Except for overloaded instance constructor declarations and static constructor
 31 declarations, a class or struct member declaration cannot introduce a member by the same name as the
 32 class or struct. A class, struct, or interface permits the declaration of overloaded methods and indexers.
 33 Furthermore, a class or struct permits the declaration of overloaded instance constructors and operators.
 34 For example, a class, struct, or interface may contain multiple method declarations with the same name,
 35 provided these method declarations differ in their signature (§10.6). Note that base classes do not
 36 contribute to the declaration space of a class, and base interfaces do not contribute to the declaration
 37 space of an interface. Thus, a derived class or interface is allowed to declare a member with the same
 38 name as an inherited member. Such a member is said to *hide* the inherited member.
- 39 • Each enumeration declaration creates a new declaration space. Names are introduced into this
 40 declaration space through *enum-member-declarations*.
- 41 • Each *block* or *switch-block* creates a different declaration space for local variables. Names are
 42 introduced into this declaration space through *local-variable-declarations*. If a block is the body of an
 43 instance constructor, method, or operator declaration, or a get or set accessor for an indexer declaration,
 44 the parameters declared in such a declaration are members of the block's *local variable declaration*
 45 *space*. The local variable declaration space of a block includes any nested blocks. Thus, within a nested
 46 block it is not possible to declare a local variable with the same name as a local variable in an enclosing
 47 block.

- 1 • Each *block* or *switch-block* creates a separate declaration space for labels. Names are introduced into
 2 this declaration space through *labeled-statements*, and the names are referenced through *goto-*
 3 *statements*. The **label declaration space** of a block includes any nested blocks. Thus, within a nested
 4 block it is not possible to declare a label with the same name as a label in an enclosing block.

5 The textual order in which names are declared is generally of no significance. In particular, textual order is
 6 not significant for the declaration and use of namespaces, constants, methods, properties, events, indexers,
 7 operators, instance constructors, destructors, static constructors, and types. Declaration order is significant in
 8 the following ways:

- 9 • Declaration order for field declarations and local variable declarations determines the order in which
 10 their initializers (if any) are executed.
- 11 • Local variables must be defined before they are used (§10.7).
- 12 • Declaration order for enum member declarations (§21.3) is significant when *constant-expression* values
 13 are omitted.

14 [*Example:* The declaration space of a namespace is “open ended”, and two namespace declarations with the
 15 same fully qualified name contribute to the same declaration space. For example

```

16     namespace Megacorp.Data
17     {
18         class Customer
19         {
20             ...
21         }
22     }
23
24     namespace Megacorp.Data
25     {
26         class Order
27         {
28             ...
29         }
  
```

30 The two namespace declarations above contribute to the same declaration space, in this case declaring two
 31 classes with the fully qualified names `Megacorp.Data.Customer` and `Megacorp.Data.Order`. Because
 32 the two declarations contribute to the same declaration space, it would have caused a compile-time error if
 33 each contained a declaration of a class with the same name. *end example*]

34 [*Note:* As specified above, the declaration space of a block includes any nested blocks. Thus, in the
 35 following example, the F and G methods result in a compile-time error because the name `i` is declared in the
 36 outer block and cannot be redeclared in the inner block. However, the H and I methods are valid since the
 37 two `i`'s are declared in separate non-nested blocks.

```

38     class A
39     {
40         void F() {
41             int i = 0;
42             if (true) {
43                 int i = 1;
44             }
45         }
46
47         void G() {
48             if (true) {
49                 int i = 0;
50             }
51             int i = 1;
  
```

```

1      void H() {
2          if (true) {
3              int i = 0;
4          }
5          if (true) {
6              int i = 1;
7          }
8      }
9
10     void I() {
11         for (int i = 0; i < 10; i++)
12             H();
13         for (int i = 0; i < 10; i++)
14             H();
15     }

```

16 *end note]*

17 10.4 Members

18 Namespaces and types have *members*. [*Note*: The members of an entity are generally available through the
19 use of a qualified name that starts with a reference to the entity, followed by a “.” token, followed by the
20 name of the member. *end note]*

21 Members of a type are either declared in the type or *inherited* from the base class of the type. When a type
22 inherits from a base class, all members of the base class, except instance constructors, destructors, and static
23 constructors become members of the derived type. The declared accessibility of a base class member does
24 not control whether the member is inherited—inheritance extends to any member that isn’t an instance
25 constructor, static constructor, or destructor. However, an inherited member may not be accessible in a
26 derived type, either because of its declared accessibility (§10.5.1) or because it is hidden by a declaration in
27 the type itself (§10.7.1.2).

28 10.4.1 Namespace members

29 Namespaces and types that have no enclosing namespace are members of the *global namespace*. This
30 corresponds directly to the names declared in the global declaration space.

31 Namespaces and types declared within a namespace are members of that namespace. This corresponds
32 directly to the names declared in the declaration space of the namespace.

33 Namespaces have no access restrictions. It is not possible to declare private, protected, or internal
34 namespaces, and namespace names are always publicly accessible.

35 10.4.2 Struct members

36 The members of a struct are the members declared in the struct and the members inherited from class
37 object.

38 The members of a simple type correspond directly to the members of the struct type aliased by the simple
39 type:

- 40 • The members of `sbyte` are the members of the `System.SByte` struct.
- 41 • The members of `byte` are the members of the `System.Byte` struct.
- 42 • The members of `short` are the members of the `System.Int16` struct.
- 43 • The members of `ushort` are the members of the `System.UInt16` struct.
- 44 • The members of `int` are the members of the `System.Int32` struct.
- 45 • The members of `uint` are the members of the `System.UInt32` struct.
- 46 • The members of `long` are the members of the `System.Int64` struct.

- 1 • The members of `ulong` are the members of the `System.UInt64` struct.
- 2 • The members of `char` are the members of the `System.Char` struct.
- 3 • The members of `float` are the members of the `System.Single` struct.
- 4 • The members of `double` are the members of the `System.Double` struct.
- 5 • The members of `decimal` are the members of the `System.Decimal` struct.
- 6 • The members of `bool` are the members of the `System.Boolean` struct.

7 **10.4.3 Enumeration members**

8 The members of an enumeration are the constants declared in the enumeration and the members inherited
9 from class `object`.

10 **10.4.4 Class members**

11 The members of a class are the members declared in the class and the members inherited from the base class
12 (except for class `object` which has no base class). The members inherited from the base class include the
13 constants, fields, methods, properties, events, indexers, operators, and types of the base class, but not the
14 instance constructors, destructors, and static constructors of the base class. Base class members are inherited
15 without regard to their accessibility.

16 A class declaration may contain declarations of constants, fields, methods, properties, events, indexers,
17 operators, instance constructors, destructors, static constructors, and types.

18 The members of `object` and `string` correspond directly to the members of the class types they alias:

- 19 • The members of `object` are the members of the `System.Object` class.
- 20 • The members of `string` are the members of the `System.String` class.

21 **10.4.5 Interface members**

22 The members of an interface are the members declared in the interface and in all base interfaces of the
23 interface, and the members inherited from class `object`.

24 **10.4.6 Array members**

25 The members of an array are the members inherited from class `System.Array`.

26 **10.4.7 Delegate members**

27 The members of a delegate are the members inherited from class `System.Delegate`.

28 **10.5 Member access**

29 Declarations of members allow control over member access. The accessibility of a member is established by
30 the declared accessibility (§10.5.1) of the member combined with the accessibility of the immediately
31 containing type, if any.

32 When access to a particular member is allowed, the member is said to be *accessible*. Conversely, when
33 access to a particular member is disallowed, the member is said to be *inaccessible*. Access to a member is
34 permitted when the textual location in which the access takes place is included in the accessibility domain
35 (§10.5.2) of the member.

36 **10.5.1 Declared accessibility**

37 The *declared accessibility* of a member can be one of the following:

- 38 • Public, which is selected by including a `public` modifier in the member declaration. The intuitive
39 meaning of `public` is “access not limited”.

- 1 • Protected, which is selected by including a `protected` modifier in the member declaration. The
2 intuitive meaning of `protected` is “access limited to the containing class or types derived from the
3 containing class”.
- 4 • Internal, which is selected by including an `internal` modifier in the member declaration. The intuitive
5 meaning of `internal` is “access limited to this program”.
- 6 • Protected internal, which is selected by including both a `protected` and an `internal` modifier in the
7 member declaration. The intuitive meaning of `protected internal` is “access limited to this program
8 or types derived from the containing class”.
- 9 • Private, which is selected by including a `private` modifier in the member declaration. The intuitive
10 meaning of `private` is “access limited to the containing type”.

11 Depending on the context in which a member declaration takes place, only certain types of declared
12 accessibility are permitted. Furthermore, when a member declaration does not include any access modifiers,
13 the context in which the declaration takes place determines the default declared accessibility.

- 14 • Namespaces implicitly have `public` declared accessibility. No access modifiers are allowed on
15 namespace declarations.
- 16 • Types declared in compilation units or namespaces can have `public` or `internal` declared
17 accessibility and default to `internal` declared accessibility.
- 18 • Class members can have any of the five kinds of declared accessibility and default to `private` declared
19 accessibility. (Note that a type declared as a member of a class can have any of the five kinds of declared
20 accessibility, whereas a type declared as a member of a namespace can have only `public` or `internal`
21 declared accessibility.)
- 22 • Struct members can have `public`, `internal`, or `private` declared accessibility and default to
23 `private` declared accessibility because structs are implicitly sealed. Struct members introduced in a
24 struct (that is, not inherited by that struct) cannot have `protected` or `protected internal` declared
25 accessibility. (Note that a type declared as a member of a struct can have `public`, `internal`, or
26 `private` declared accessibility, whereas a type declared as a member of a namespace can have only
27 `public` or `internal` declared accessibility.)
- 28 • Interface members implicitly have `public` declared accessibility. No access modifiers are allowed on
29 interface member declarations.
- 30 • Enumeration members implicitly have `public` declared accessibility. No access modifiers are allowed
31 on enumeration member declarations.

32 10.5.2 Accessibility domains

33 The *accessibility domain* of a member consists of the (possibly disjoint) sections of program text in which
34 access to the member is permitted. For purposes of defining the accessibility domain of a member, a member
35 is said to be *top-level* if it is not declared within a type, and a member is said to be *nested* if it is declared
36 within another type. Furthermore, the text of an assembly is defined as all source text contained in all source
37 files of that assembly, and the source text of a type is defined as all source text contained between the
38 opening and closing “{” and “}” tokens in the *class-body*, *struct-body*, *interface-body*, or *enum-body* of the
39 type (including, possibly, types that are nested within the type).

40 The accessibility domain of a predefined type (such as `object`, `int`, or `double`) is unlimited.

41 The accessibility domain of a top-level type `T` that is declared in a program `P` is defined as follows:

- 42 • If the declared accessibility of `T` is `public`, the accessibility domain of `T` is the program text of `P` and
43 any program that references `P`.
- 44 • If the declared accessibility of `T` is `internal`, the accessibility domain of `T` is the program text of `P`.

1 [Note: From these definitions it follows that the accessibility domain of a top-level type is always at least the
2 program text of the program in which that type is declared. *end note*]

3 The accessibility domain of a nested member M declared in a type T within a program P, is defined as
4 follows (noting that M itself may possibly be a type):

- 5 • If the declared accessibility of M is `public`, the accessibility domain of M is the accessibility domain
6 of T.
- 7 • If the declared accessibility of M is `protected internal`, let D be the union of the program text of P
8 and the program text of any type derived from T, which is declared outside P. The accessibility domain
9 of M is the intersection of the accessibility domain of T with D.
- 10 • If the declared accessibility of M is `protected`, let D be the union of the program text of T and the
11 program text of any type derived from T. The accessibility domain of M is the intersection of the
12 accessibility domain of T with D.
- 13 • If the declared accessibility of M is `internal`, the accessibility domain of M is the intersection of the
14 accessibility domain of T with the program text of P.
- 15 • If the declared accessibility of M is `private`, the accessibility domain of M is the program text of T.

16 [Note: From these definitions it follows that the accessibility domain of a nested member is always at least
17 the program text of the type in which the member is declared. Furthermore, it follows that the accessibility
18 domain of a member is never more inclusive than the accessibility domain of the type in which the member
19 is declared. *end note*]

20 [Note: In intuitive terms, when a type or member M is accessed, the following steps are evaluated to ensure
21 that the access is permitted:

- 22 • First, if M is declared within a type (as opposed to a compilation unit or a namespace), a compile-time
23 error occurs if that type is not accessible.
- 24 • Then, if M is `public`, the access is permitted.
- 25 • Otherwise, if M is `protected internal`, the access is permitted if it occurs within the program in
26 which M is declared, or if it occurs within a class derived from the class in which M is declared and takes
27 place through the derived class type (§10.5.3).
- 28 • Otherwise, if M is `protected`, the access is permitted if it occurs within the class in which M is declared,
29 or if it occurs within a class derived from the class in which M is declared and takes place through the
30 derived class type (§10.5.3).
- 31 • Otherwise, if M is `internal`, the access is permitted if it occurs within the program in which M is
32 declared.
- 33 • Otherwise, if M is `private`, the access is permitted if it occurs within the type in which M is declared.
- 34 • Otherwise, the type or member is inaccessible, and a compile-time error occurs.

35 *end note*]

36 [Example: In the example

```
37     public class A
38     {
39         public static int X;
40         internal static int Y;
41         private static int Z;
42     }
43     internal class B
44     {
45         public static int X;
46         internal static int Y;
47         private static int Z;
```

C# LANGUAGE SPECIFICATION

```
1      public class C
2      {
3          public static int X;
4          internal static int Y;
5          private static int Z;
6      }
7
8      private class D
9      {
10         public static int X;
11         internal static int Y;
12         private static int Z;
13     }
```

14 the classes and members have the following accessibility domains:

- 15 • The accessibility domain of `A` and `A.X` is unlimited.
- 16 • The accessibility domain of `A.Y`, `B`, `B.X`, `B.Y`, `B.C`, `B.C.X`, and `B.C.Y` is the program text of the
17 containing program.
- 18 • The accessibility domain of `A.Z` is the program text of `A`.
- 19 • The accessibility domain of `B.Z` and `B.D` is the program text of `B`, including the program text of `B.C`
20 and `B.D`.
- 21 • The accessibility domain of `B.C.Z` is the program text of `B.C`.
- 22 • The accessibility domain of `B.D.X`, `B.D.Y`, and `B.D.Z` is the program text of `B.D`.

23 As the example illustrates, the accessibility domain of a member is never larger than that of a containing
24 type. For example, even though all `X` members have public declared accessibility, all but `A.X` have
25 accessibility domains that are constrained by a containing type. *end example*]

26 As described in §10.4, all members of a base class, except for instance constructors, destructors, and static
27 constructors are inherited by derived types. This includes even private members of a base class. However,
28 the accessibility domain of a private member includes only the program text of the type in which the
29 member is declared. [*Example*: In the example

```
30     class A
31     {
32         int x;
33         static void F(B b) {
34             b.x = 1;    // Ok
35         }
36     }
37
38     class B: A
39     {
40         static void F(B b) {
41             b.x = 1;    // Error, x not accessible
42         }
43     }
```

43 the `B` class inherits the private member `x` from the `A` class. Because the member is private, it is only
44 accessible within the *class-body* of `A`. Thus, the access to `b.x` succeeds in the `A.F` method, but fails in the
45 `B.F` method. *end example*]

46 10.5.3 Protected access for instance members

47 When a **protected** instance member is accessed outside the program text of the class in which it is
48 declared, and when a **protected internal** instance member is accessed outside the program text of the
49 program in which it is declared, the access is required to take place *through* an instance of the derived class
50 type in which the access occurs. Let `B` be a base class that declares a protected instance member `M`, and let `D`
51 be a class that derives from `B`. Within the *class-body* of `D`, access to `M` can take one of the following forms:

- 52 • An unqualified *type-name* or *primary-expression* of the form `M`.

- 1 • A *primary-expression* of the form `E.M`, provided the type of `E` is `D` or a class derived from `D`.
- 2 • A *primary-expression* of the form `base.M`.

3 In addition to these forms of access, a derived class can access a protected instance constructor of a base
4 class in a *constructor-initializer* (§17.10.1).

5 [*Example:* In the example

```
6     public class A
7     {
8         protected int x;
9
10        static void F(A a, B b) {
11            a.x = 1;    // Ok
12            b.x = 1;    // Ok
13        }
14
15        public class B: A
16        {
17            static void F(A a, B b) {
18                a.x = 1;    // Error, must access through instance of B
19                b.x = 1;    // Ok
20            }
21        }
22    }
```

21 within `A`, it is possible to access `x` through instances of both `A` and `B`, since in either case the access takes
22 place *through* an instance of `A` or a class derived from `A`. However, within `B`, it is not possible to access `x`
23 through an instance of `A`, since `A` does not derive from `B`. *end example*]

24 10.5.4 Accessibility constraints

25 Several constructs in the `C#` language require a type to be *at least as accessible as* a member or another type.
26 A type `T` is said to be at least as accessible as a member or type `M` if the accessibility domain of `T` is a
27 superset of the accessibility domain of `M`. In other words, `T` is at least as accessible as `M` if `T` is accessible in
28 all contexts in which `M` is accessible.

29 The following accessibility constraints exist:

- 30 • The direct base class of a class type must be at least as accessible as the class type itself.
- 31 • The explicit base interfaces of an interface type must be at least as accessible as the interface type itself.
- 32 • The return type and parameter types of a delegate type must be at least as accessible as the delegate type
33 itself.
- 34 • The type of a constant must be at least as accessible as the constant itself.
- 35 • The type of a field must be at least as accessible as the field itself.
- 36 • The return type and parameter types of a method must be at least as accessible as the method itself.
- 37 • The type of a property must be at least as accessible as the property itself.
- 38 • The type of an event must be at least as accessible as the event itself.
- 39 • The type and parameter types of an indexer must be at least as accessible as the indexer itself.
- 40 • The return type and parameter types of an operator must be at least as accessible as the operator itself.
- 41 • The parameter types of an instance constructor must be at least as accessible as the instance constructor
42 itself.

43 [*Example:* In the example

```
44     class A {...}
45     public class B: A {...}
```

1 the B class results in a compile-time error because A is not at least as accessible as B. *end example*]

2 [*Example:* Likewise, in the example

```
3     class A {...}
4     public class B
5     {
6         A F() {...}
7         internal A G() {...}
8         public A H() {...}
9     }
```

10 the H method in B results in a compile-time error because the return type A is not at least as accessible as the
11 method. *end example*]

12 10.6 Signatures and overloading

13 Methods, instance constructors, indexers, and operators are characterized by their *signatures*:

- 14 • The signature of a method consists of the name of the method and the type and kind (value, reference, or
15 output) of each of its formal parameters, considered in the order left to right. The signature of a method
16 specifically does not include the return type, nor does it include the `params` modifier that may be
17 specified for the right-most parameter.
- 18 • The signature of an instance constructor consists of the type and kind (value, reference, or output) of
19 each of its formal parameters, considered in the order left to right. The signature of an instance
20 constructor specifically does not include the `params` modifier that may be specified for the right-most
21 parameter.
- 22 • The signature of an indexer consists of the type of each of its formal parameters, considered in the order
23 left to right. The signature of an indexer specifically does not include the element type.
- 24 • The signature of an operator consists of the name of the operator and the type of each of its formal
25 parameters, considered in the order left to right. The signature of an operator specifically does not
26 include the result type.

27 Signatures are the enabling mechanism for *overloading* of members in classes, structs, and interfaces:

- 28 • Overloading of methods permits a class, struct, or interface to declare multiple methods with the same
29 name, provided their signatures are unique within that class, struct, or interface.
- 30 • Overloading of instance constructors permits a class or struct to declare multiple instance constructors,
31 provided their signatures are unique within that class or struct.
- 32 • Overloading of indexers permits a class, struct, or interface to declare multiple indexers, provided their
33 signatures are unique within that class, struct, or interface.
- 34 • Overloading of operators permits a class or struct to declare multiple operators with the same name,
35 provided their signatures are unique within that class or struct.

36 [*Example:* The following example shows a set of overloaded method declarations along with their
37 signatures.

```
38     interface ITest
39     {
40         void F(); // F()
41         void F(int x); // F(int)
42         void F(ref int x); // F(ref int)
43         void F(out int x); // F(out int)
44         void F(int x, int y); // F(int, int)
45         int F(string s); // F(string)
```

```

1      int F(int x);           // F(int)      error
2      void F(string[] a);    // F(string[])
3      void F(params string[] a); // F(string[])  error
4  }
```

5 Note that any `ref` and `out` parameter modifiers (§17.5.1) are part of a signature. Thus, `F(int)`, `F(ref`
6 `int)`, and `F(out int)` are all unique signatures. Also, note that the return type and the `params` modifier
7 are not part of a signature, so it is not possible to overload solely based on return type or on the inclusion or
8 exclusion of the `params` modifier. As such, the declarations of the methods `F(int)` and `F(params`
9 `string[])` identified above, result in a compile-time error. *end example*

10 10.7 Scopes

11 The *scope* of a name is the region of program text within which it is possible to refer to the entity declared
12 by the name without qualification of the name. Scopes can be *nested*, and an inner scope may redeclare the
13 meaning of a name from an outer scope. [*Note*: This does not, however, remove the restriction imposed by
14 §10.3 that within a nested block it is not possible to declare a local variable with the same name as a local
15 variable in an enclosing block. *end note*] The name from the outer scope is then said to be *hidden* in the
16 region of program text covered by the inner scope, and access to the outer name is only possible by
17 qualifying the name.

- 18 • The scope of a namespace member declared by a *namespace-member-declaration* (§16.4) with no
19 enclosing *namespace-declaration* is the entire program text.
- 20 • The scope of a namespace member declared by a *namespace-member-declaration* within a *namespace-*
21 *declaration* whose fully qualified name is N, is the *namespace-body* of every *namespace-declaration*
22 whose fully qualified name is N or starts with N, followed by a period.
- 23 • The scope of a name defined or imported by a *using-directive* (§16.3) extends over the *namespace-*
24 *member-declarations* of the *compilation-unit* or *namespace-body* in which the *using-directive* occurs. A
25 *using-directive* may make zero or more namespace or type names available within a particular
26 *compilation-unit* or *namespace-body*, but does not contribute any new members to the underlying
27 declaration space. In other words, a *using-directive* is not transitive, but, rather, affects only the
28 *compilation-unit* or *namespace-body* in which it occurs.
- 29 • The scope of a member declared by a *class-member-declaration* (§17.2) is the *class-body* in which the
30 declaration occurs. In addition, the scope of a class member extends to the *class-body* of those derived
31 classes that are included in the accessibility domain (§10.5.2) of the member.
- 32 • The scope of a member declared by a *struct-member-declaration* (§18.2) is the *struct-body* in which the
33 declaration occurs.
- 34 • The scope of a member declared by an *enum-member-declaration* (§21.3) is the *enum-body* in which the
35 declaration occurs.
- 36 • The scope of a parameter declared in a *method-declaration* (§17.5) is the *method-body* of that *method-*
37 *declaration*.
- 38 • The scope of a parameter declared in an *indexer-declaration* (§17.8) is the *accessor-declarations* of that
39 *indexer-declaration*.
- 40 • The scope of a parameter declared in an *operator-declaration* (§17.9) is the *block* of that *operator-*
41 *declaration*.
- 42 • The scope of a parameter declared in a *constructor-declaration* (§17.10) is the *constructor-initializer*
43 and *block* of that *constructor-declaration*.
- 44 • The scope of a label declared in a *labeled-statement* (§15.4) is the *block* in which the declaration occurs.
- 45 • The scope of a local variable declared in a *local-variable-declaration* (§15.5.1) is the *block* in which the
46 declaration occurs.

C# LANGUAGE SPECIFICATION

- 1 • The scope of a local variable declared in a *switch-block* of a `switch` statement (§15.7.2) is the *switch-*
2 *block*.
- 3 • The scope of a local variable declared in a *for-initializer* of a `for` statement (§15.8.3) is the *for-*
4 *initializer*, the *for-condition*, the *for-iterator*, and the contained *statement* of the `for` statement.
- 5 • The scope of a local constant declared in a *local-constant-declaration* (§15.5.2) is the block in which the
6 declaration occurs. It is a compile-time error to refer to a local constant in a textual position that
7 precedes its *constant-declarator*.

8 Within the scope of a namespace, class, struct, or enumeration member it is possible to refer to the member
9 in a textual position that precedes the declaration of the member. [Example: For example

```
10     class A  
11     {  
12         void F() {  
13             i = 1;  
14         }  
15         int i = 0;  
16     }
```

17 Here, it is valid for `F` to refer to `i` before it is declared. *end example*]

18 Within the scope of a local variable, it is a compile-time error to refer to the local variable in a textual
19 position that precedes the *local-variable-declarator* of the local variable. [Example: For example

```
20     class A  
21     {  
22         int i = 0;  
23         void F() {  
24             i = 1;           // Error, use precedes declaration  
25             int i;  
26             i = 2;  
27         }  
28         void G() {  
29             int j = (j = 1); // valid  
30         }  
31         void H() {  
32             int a = 1, b = ++a; // valid  
33         }  
34     }
```

35 In the `F` method above, the first assignment to `i` specifically does not refer to the field declared in the outer
36 scope. Rather, it refers to the local variable and it results in a compile-time error because it textually
37 precedes the declaration of the variable. In the `G` method, the use of `j` in the initializer for the declaration of
38 `j` is valid because the use does not precede the *local-variable-declarator*. In the `H` method, a subsequent
39 *local-variable-declarator* correctly refers to a local variable declared in an earlier *local-variable-declarator*
40 within the same *local-variable-declaration*. *end example*]

41 [Note: The scoping rules for local variables are designed to guarantee that the meaning of a name used in an
42 expression context is always the same within a block. If the scope of a local variable were to extend only
43 from its declaration to the end of the block, then in the example above, the first assignment would assign to
44 the instance variable and the second assignment would assign to the local variable, possibly leading to
45 compile-time errors if the statements of the block were later to be rearranged.

46 The meaning of a name within a block may differ based on the context in which the name is used. In the
47 example

```
48     using System;  
49     class A {}
```

```

1      class Test
2      {
3          static void Main() {
4              string A = "hello, world";
5              string s = A;                // expression context
6              Type t = typeof(A);         // type context
7              Console.WriteLine(s);       // writes "hello, world"
8              Console.WriteLine(t.ToString()); // writes "Type: A"
9          }
10     }

```

11 the name `A` is used in an expression context to refer to the local variable `A` and in a type context to refer to
 12 the class `A`. *end note*

13 10.7.1 Name hiding

14 The scope of an entity typically encompasses more program text than the declaration space of the entity. In
 15 particular, the scope of an entity may include declarations that introduce new declaration spaces containing
 16 entities of the same name. Such declarations cause the original entity to become *hidden*. Conversely, an
 17 entity is said to be *visible* when it is not hidden.

18 Name hiding occurs when scopes overlap through nesting and when scopes overlap through inheritance. The
 19 characteristics of the two types of hiding are described in the following sections.

20 10.7.1.1 Hiding through nesting

21 Name hiding through nesting can occur as a result of nesting namespaces or types within namespaces, as a
 22 result of nesting types within classes or structs, and as a result of parameter and local variable declarations.

23 [*Example*: In the example

```

24     class A
25     {
26         int i = 0;
27         void F() {
28             int i = 1;
29         }
30         void G() {
31             i = 1;
32         }
33     }

```

34 within the `F` method, the instance variable `i` is hidden by the local variable `i`, but within the `G` method, `i` still
 35 refers to the instance variable. *end example*

36 When a name in an inner scope hides a name in an outer scope, it hides all overloaded occurrences of that
 37 name. [*Example*: In the example

```

38     class Outer
39     {
40         static void F(int i) {}
41         static void F(string s) {}
42         class Inner
43         {
44             void G() {
45                 F(1);           // Invokes Outer.Inner.F
46                 F("Hello");    // Error
47             }
48             static void F(long l) {}
49         }
50     }

```

51 the call `F(1)` invokes the `F` declared in `Inner` because all outer occurrences of `F` are hidden by the inner
 52 declaration. For the same reason, the call `F("Hello")` results in a compile-time error. *end example*

1 10.7.1.2 Hiding through inheritance

2 Name hiding through inheritance occurs when classes or structs redeclare names that were inherited from
3 base classes. This type of name hiding takes one of the following forms:

- 4 • A constant, field, property, event, or type introduced in a class or struct hides all base class members
5 with the same name.
- 6 • A method introduced in a class or struct hides all non-method base class members with the same name,
7 and all base class methods with the same signature (method name and parameter count, modifiers, and
8 types).
- 9 • An indexer introduced in a class or struct hides all base class indexers with the same signature
10 (parameter count and types).

11 The rules governing operator declarations (§17.9) make it impossible for a derived class to declare an
12 operator with the same signature as an operator in a base class. Thus, operators never hide one another.

13 Contrary to hiding a name from an outer scope, hiding an accessible name from an inherited scope causes a
14 warning to be reported. [*Example*: In the example

```
15     class Base
16     {
17         public void F() {}
18     }
19     class Derived: Base
20     {
21         public void F() {}      // warning, hiding an inherited name
22     }
```

23 the declaration of `F` in `Derived` causes a warning to be reported. Hiding an inherited name is specifically
24 not an error, since that would preclude separate evolution of base classes. For example, the above situation
25 might have come about because a later version of `Base` introduced an `F` method that wasn't present in an
26 earlier version of the class. Had the above situation been an error, then *any* change made to a base class in a
27 separately versioned class library could potentially cause derived classes to become invalid. *end example*]

28 The warning caused by hiding an inherited name can be eliminated through use of the `new` modifier:
29 [*Example*:

```
30     class Base
31     {
32         public void F() {}
33     }
34     class Derived: Base
35     {
36         new public void F() {}
37     }
```

38 The `new` modifier indicates that the `F` in `Derived` is “new”, and that it is indeed intended to hide the
39 inherited member. *end example*]

40 A declaration of a new member hides an inherited member only within the scope of the new member.

41 [*Example*:

```
42     class Base
43     {
44         public static void F() {}
45     }
46     class Derived: Base
47     {
48         new private static void F() {}    // Hides Base.F in Derived only
49     }
```



```

1      class MoreDerived: Derived
2      {
3          static void G() { F(); }           // Invokes Base.F
4      }

```

5 In the example above, the declaration of `F` in `Derived` hides the `F` that was inherited from `Base`, but since
6 the new `F` in `Derived` has private access, its scope does not extend to `MoreDerived`. Thus, the call `F()` in
7 `MoreDerived.G` is valid and will invoke `Base.F`. *end example*]

8 10.8 Namespace and type names

9 Several contexts in a C# program require a *namespace-name* or a *type-name* to be specified. Either form of
10 name is written as one or more identifiers separated by “.” tokens.

```

11      namespace-name:
12          namespace-or-type-name
13
14      type-name:
15          namespace-or-type-name
16
17      namespace-or-type-name:
18          identifier
19          namespace-or-type-name . identifier

```

18 A *type-name* is a *namespace-or-type-name* that refers to a type. Following resolution as described below, the
19 *namespace-or-type-name* of a *type-name* must refer to a type, or otherwise a compile-time error occurs.

20 A *namespace-name* is a *namespace-or-type-name* that refers to a namespace. Following resolution as
21 described below, the *namespace-or-type-name* of a *namespace-name* must refer to a namespace, or
22 otherwise a compile-time error occurs.

23 The meaning of a *namespace-or-type-name* is determined as follows:

- 24 • If the *namespace-or-type-name* consists of a single identifier:
 - 25 ○ If the *namespace-or-type-name* appears within the body of a class or struct declaration, then starting
26 with that class or struct declaration and continuing with each enclosing class or struct declaration (if
27 any), if a member with the given name exists, is accessible, and denotes a type, then the *namespace-
28 or-type-name* refers to that member. Note that non-type members (constants, fields, methods,
29 properties, indexers, operators, instance constructors, destructors, and static constructors) are
30 ignored when determining the meaning of a *namespace-or-type-name*.
 - 31 ○ Otherwise, starting with the namespace in which the *namespace-or-type-name* occurs, continuing
32 with each enclosing namespace (if any), and ending with the global namespace, the following steps
33 are evaluated until an entity is located:
 - 34 • If the namespace contains a namespace member with the given name, then the *namespace-or-
35 type-name* refers to that member and, depending on the member, is classified as a namespace or
36 a type.
 - 37 • Otherwise, if the namespace has a corresponding namespace declaration enclosing the location
38 where the *namespace-or-type-name* occurs, then:
 - 39 ○ If the namespace declaration contains a using-alias-directive that associates the given name
40 with an imported namespace or type, then the *namespace-or-type-name* refers to that
41 namespace or type.
 - 42 ○ Otherwise, if the namespaces imported by the using-namespace-directives of the namespace
43 declaration contain exactly one type with the given name, then the *namespace-or-type-name*
44 refers to that type.
 - 45 ○ Otherwise, if the namespaces imported by the using-namespace-directives of the namespace
46 declaration contain more than one type with the given name, then the *namespace-or-type-
47 name* is ambiguous and an error occurs.

- 1 ○ Otherwise, the *namespace-or-type-name* is undefined and a compile-time error occurs.
- 2 • Otherwise, the *namespace-or-type-name* is of the form *N . I*, where *N* is a *namespace-or-type-name*
- 3 consisting of all identifiers but the rightmost one, and *I* is the rightmost identifier. *N* is first resolved as a
- 4 *namespace-or-type-name*. If the resolution of *N* is not successful, a compile-time error occurs.
- 5 Otherwise, *N . I* is resolved as follows:
- 6 ○ If *N* is a namespace and *I* is the name of an accessible member of that namespace, then *N . I* refers to
- 7 that member and, depending on the member, is classified as a namespace or a type.
- 8 ○ If *N* is a class or struct type and *I* is the name of an accessible type in *N*, then *N . I* refers to that type.
- 9 ○ Otherwise, *N . I* is an *invalid namespace-or-type-name*, and a compile-time error occurs.

10 10.8.1 Fully qualified names

11 Every namespace and type has a *fully qualified name*, which uniquely identifies the namespace or type

12 amongst all others. The fully qualified name of a namespace or type *N* is determined as follows:

- 13 • If *N* is a member of the global namespace, its fully qualified name is *N*.
- 14 • Otherwise, its fully qualified name is *S . N*, where *S* is the fully qualified name of the namespace or type
- 15 in which *N* is declared.

16 In other words, the fully qualified name of *N* is the complete hierarchical path of identifiers that lead to *N*,

17 starting from the global namespace. Because every member of a namespace or type must have a unique

18 name, it follows that the fully qualified name of a namespace or type is always unique.

19 [*Example:* The example below shows several namespace and type declarations along with their associated

20 fully qualified names.

```

21     class A {}           // A
22     namespace X         // X
23     {
24         class B         // X.B
25         {
26             class C {}  // X.B.C
27         }
28         namespace Y     // X.Y
29         {
30             class D {}  // X.Y.D
31         }
32     }
33     namespace X.Y       // X.Y
34     {
35         class E {}      // X.Y.E
36     }

```

37 *end example*]

38 10.9 Automatic memory management

39 C# employs automatic memory management, which frees developers from manually allocating and freeing

40 the memory occupied by objects. Automatic memory management policies are implemented by a garbage

41 collector. The memory management life cycle of an object is as follows:

- 42 1. When the object is created, memory is allocated for it, the constructor is run, and the object is
- 43 considered *live*.
- 44 2. If the object, or any part of it, cannot be accessed by any possible continuation of execution, other than
- 45 the running of destructors, the object is considered *no longer in use*, and it becomes eligible for
- 46 destruction. [*Note:* Implementations may choose to analyze code to determine which references to an
- 47 object may be used in the future. For instance, if a local variable that is in scope is the only existing
- 48 reference to an object, but that local variable is never referred to in any possible continuation of

- 1 execution from the current execution point in the procedure, an implementation may (but is not required to) treat the object as no longer in use. *end note*
- 2
- 3 3. Once the object is eligible for destruction, at some unspecified later time the destructor (§17.12) (if any)
 - 4 for the object is run. Unless overridden by explicit calls, the destructor for the object is run once only.
 - 5 4. Once the destructor for an object is run, if that object, or any part of it, cannot be accessed by any
 - 6 possible continuation of execution, including the running of destructors, the object is considered
 - 7 **inaccessible** and the object becomes eligible for collection.
 - 8 5. Finally, at some time after the object becomes eligible for collection, the garbage collector frees the
 - 9 memory associated with that object.

10 The garbage collector maintains information about object usage, and uses this information to make memory

11 management decisions, such as where in memory to locate a newly created object, when to relocate an

12 object, and when an object is no longer in use or inaccessible.

13 Like other languages that assume the existence of a garbage collector, C# is designed so that the garbage

14 collector may implement a wide range of memory management policies. For instance, C# does not require

15 that destructors be run or that objects be collected as soon as they are eligible, or that destructors be run in

16 any particular order, or on any particular thread.

17 The behavior of the garbage collector can be controlled, to some degree, via static methods on the class

18 `System.GC`. This class can be used to request a collection to occur, destructors to be run (or not run), and so

19 forth.

20 [*Example:* Since the garbage collector is allowed wide latitude in deciding when to collect objects and run

21 destructors, a conforming implementation may produce output that differs from that shown by the following

22 code. The program

```

23     using System;
24     class A
25     {
26         ~A() {
27             Console.WriteLine("Destruct instance of A");
28         }
29     }
30     class B
31     {
32         object Ref;
33         public B(object o) {
34             Ref = o;
35         }
36         ~B() {
37             Console.WriteLine("Destruct instance of B");
38         }
39     }
40     class Test
41     {
42         static void Main() {
43             B b = new B(new A());
44             b = null;
45             GC.Collect();
46             GC.WaitForPendingFinalizers();
47         }
48     }

```

49 creates an instance of class A and an instance of class B. These objects become eligible for garbage

50 collection when the variable `b` is assigned the value `null`, since after this time it is impossible for any user-

51 written code to access them. The output could be either

```

52     Destruct instance of A
53     Destruct instance of B

```

54 or

C# LANGUAGE SPECIFICATION

1 Destruct instance of B
2 Destruct instance of A

3 because the language imposes no constraints on the order in which objects are garbage collected.

4 In subtle cases, the distinction between “eligible for destruction” and “eligible for collection” can be
5 important. For example,

```
6       using System;
7       class A
8       {
9           ~A() {
10              Console.WriteLine("Destruct instance of A");
11           }
12           public void F() {
13              Console.WriteLine("A.F");
14              Test.RefA = this;
15           }
16       }
17       class B
18       {
19           public A Ref;
20           ~B() {
21              Console.WriteLine("Destruct instance of B");
22              Ref.F();
23           }
24       }
25       class Test
26       {
27           public static A RefA;
28           public static B RefB;
29           static void Main() {
30              RefB = new B();
31              RefA = new A();
32              RefB.Ref = RefA;
33              RefB = null;
34              RefA = null;
35              // A and B now eligible for destruction
36              GC.Collect();
37              GC.WaitForPendingFinalizers();
38              // B now eligible for collection, but A is not
39              if (RefA != null)
40                  Console.WriteLine("RefA is not null");
41           }
42       }
```

43 In the above program, if the garbage collector chooses to run the destructor of B before the destructor of A,
44 then the output of this program might be:

```
45       Destruct instance of A
46       Destruct instance of B
47       A.F
48       RefA is not null
```

49 Note that although the instance of A was not in use and A's destructor was run, it is still possible for methods
50 of A (in this case, F) to be called from another destructor. Also, note that running of a destructor may cause
51 an object to become usable from the mainline program again. In this case, the running of B's destructor
52 caused an instance of A that was previously not in use to become accessible from the live reference RefA.
53 After the call to `waitForPendingFinalizers`, the instance of B is eligible for collection, but the instance
54 of A is not, because of the reference RefA.

55 To avoid confusion and unexpected behavior, it is generally a good idea for destructors to only perform
56 cleanup on data stored in their object's own fields, and not to perform any actions on referenced objects or
57 static fields. *end example*]

1 10.10 Execution order

2 Execution shall proceed such that the side effects of each executing thread are preserved at critical execution
3 points. A *side effect* is defined as a read or write of a volatile field, a write to a non-volatile variable, a write
4 to an external resource, and the throwing of an exception. The critical execution points at which the order of
5 these side effects must be preserved are references to volatile fields (§17.4.3), lock statements (§15.12), and
6 thread creation and termination. An implementation is free to change the order of execution of a
7 C# program, subject to the following constraints:

- 8 • Data dependence is preserved within a thread of execution. That is, the value of each variable is
9 computed as if all statements in the thread were executed in original program order.
- 10 • Initialization ordering rules are preserved (§17.4.4 and §17.4.5).
- 11 • The ordering of side effects is preserved with respect to volatile reads and writes (§17.4.3). Additionally,
12 an implementation need not evaluate part of an expression if it can deduce that that expression's value is
13 not used and that no needed side effects are produced (including any caused by calling a method or
14 accessing a volatile field). When program execution is interrupted by an asynchronous event (such as an
15 exception thrown by another thread), it is not guaranteed that the observable side effects are visible in
16 the original program order.

17

11. Types

1

2 The types of the C# language are divided into two main categories: Value types and reference types.

3 *type:*
 4 *value-type*
 5 *reference-type*

6 A third category of types, pointers, is available only in unsafe code. This is discussed further in §25.2.

7 Value types differ from reference types in that variables of the value types directly contain their data,
 8 whereas variables of the reference types store *references* to their data, the latter being known as *objects*.
 9 With reference types, it is possible for two variables to reference the same object, and thus possible for
 10 operations on one variable to affect the object referenced by the other variable. With value types, the
 11 variables each have their own copy of the data, and it is not possible for operations on one to affect the other.

12 C#'s type system is unified such that *a value of any type can be treated as an object*. Every type in C#
 13 directly or indirectly derives from the **object** class type, and **object** is the ultimate base class of all types.
 14 Values of reference types are treated as objects simply by viewing the values as type **object**. Values of
 15 value types are treated as objects by performing boxing and unboxing operations (§11.3).

16 11.1 Value types

17 A value type is either a struct type or an enumeration type. C# provides a set of predefined struct types
 18 called the *simple types*. The simple types are identified through reserved words.

19 *value-type:*
 20 *struct-type*
 21 *enum-type*

22 *struct-type:*
 23 *type-name*
 24 *simple-type*

25 *simple-type:*
 26 *numeric-type*
 27 **bool**

28 *numeric-type:*
 29 *integral-type*
 30 *floating-point-type*
 31 **decimal**

32 *integral-type:*
 33 **sbyte**
 34 **byte**
 35 **short**
 36 **ushort**
 37 **int**
 38 **uint**
 39 **long**
 40 **ulong**
 41 **char**

1 *floating-point-type:*

2 float

3 double

4 *enum-type:*

5 *type-name*

6 All value types implicitly inherit from class `object`. It is not possible for any type to derive from a value
7 type, and value types are thus implicitly sealed (§17.1.1.2).

8 A variable of a value type always contains a value of that type. Unlike reference types, it is not possible for
9 a value of a value type to be `null`, or to reference an object of a more derived type.

10 Assignment to a variable of a value type creates a *copy* of the value being assigned. This differs from
11 assignment to a variable of a reference type, which copies the reference but not the object identified by the
12 reference.

13 11.1.1 Default constructors

14 All value types implicitly declare a public parameterless instance constructor called the *default constructor*.
15 The default constructor returns a zero-initialized instance known as the *default value* for the value type:

- 16 • For all *simple-types*, the default value is the value produced by a bit pattern of all zeros:
 - 17 ○ For `sbyte`, `byte`, `short`, `ushort`, `int`, `uint`, `long`, and `ulong`, the default value is 0.
 - 18 ○ For `char`, the default value is `'\x0000'`.
 - 19 ○ For `float`, the default value is `0.0f`.
 - 20 ○ For `double`, the default value is `0.0d`.
 - 21 ○ For `decimal`, the default value is `0.0m`.
 - 22 ○ For `bool`, the default value is `false`.
- 23 • For an *enum-type* `E`, the default value is 0.
- 24 • For a *struct-type*, the default value is the value produced by setting all value type fields to their default
25 value and all reference type fields to `null`.

26 Like any other instance constructor, the default constructor of a value type is invoked using the `new`
27 operator. [*Note:* For efficiency reasons, this requirement is not intended to actually have the implementation
28 generate a constructor call. *end note*] In the example below, variables `i` and `j` are both initialized to zero.

```
29     class A
30     {
31         void F() {
32             int i = 0;
33             int j = new int();
34         }
35     }
```

36 Because every value type implicitly has a public parameterless instance constructor, it is not possible for a
37 struct type to contain an explicit declaration of a parameterless constructor. A struct type is however
38 permitted to declare parameterized instance constructors (§18.3.8).

39 11.1.2 Struct types

40 A struct type is a value type that can declare constants, fields, methods, properties, indexers, operators,
41 instance constructors, static constructors, and nested types. Struct types are described in §18.

11.1.3 Simple types

C# provides a set of predefined struct types called the simple types. The simple types are identified through reserved words, but these reserved words are simply aliases for predefined struct types in the `System` namespace, as described in the table below.

Reserved word	Aliased type
<code>sbyte</code>	<code>System.SByte</code>
<code>byte</code>	<code>System.Byte</code>
<code>short</code>	<code>System.Int16</code>
<code>ushort</code>	<code>System.UInt16</code>
<code>int</code>	<code>System.Int32</code>
<code>uint</code>	<code>System.UInt32</code>
<code>long</code>	<code>System.Int64</code>
<code>ulong</code>	<code>System.UInt64</code>
<code>char</code>	<code>System.Char</code>
<code>float</code>	<code>System.Single</code>
<code>double</code>	<code>System.Double</code>
<code>bool</code>	<code>System.Boolean</code>
<code>decimal</code>	<code>System.Decimal</code>

Because a simple type aliases a struct type, every simple type has members. For example, `int` has the members declared in `System.Int32` and the members inherited from `System.Object`, and the following statements are permitted:

```
int i = int.MaxValue;           // System.Int32.MaxValue constant
string s = i.ToString();       // System.Int32.ToString() instance method
string t = 123.ToString();     // System.Int32.ToString() instance method
```

The simple types differ from other struct types in that they permit certain additional operations:

- Most simple types permit values to be created by writing *literals* (§9.4.4). For example, `123` is a literal of type `int` and `'a'` is a literal of type `char`. C# makes no provision for literals of struct types in general, and non-default values of other struct types are ultimately always created through instance constructors of those struct types.
- When the operands of an expression are all simple type constants, it is possible for the compiler to evaluate the expression at compile-time. Such an expression is known as a *constant-expression* (§14.15). Expressions involving operators defined by other struct types are not considered to be constant expressions.
- Through `const` declarations, it is possible to declare constants of the simple types (§17.3). It is not possible to have constants of other struct types, but a similar effect is provided by `static readonly` fields.
- Conversions involving simple types can participate in evaluation of conversion operators defined by other struct types, but a user-defined conversion operator can never participate in evaluation of another user-defined operator (§13.4.2).

11.1.4 Integral types

C# supports nine integral types: `sbyte`, `byte`, `short`, `ushort`, `int`, `uint`, `long`, `ulong`, and `char`. The integral types have the following sizes and ranges of values:

- The `sbyte` type represents signed 8-bit integers with values between `-128` and `127`.

C# LANGUAGE SPECIFICATION

- 1 • The `byte` type represents unsigned 8-bit integers with values between 0 and 255.
- 2 • The `short` type represents signed 16-bit integers with values between -32768 and 32767 .
- 3 • The `ushort` type represents unsigned 16-bit integers with values between 0 and 65535.
- 4 • The `int` type represents signed 32-bit integers with values between -2147483648 and 2147483647 .
- 5 • The `uint` type represents unsigned 32-bit integers with values between 0 and 4294967295.
- 6 • The `long` type represents signed 64-bit integers with values between -9223372036854775808 and
7 9223372036854775807 .
- 8 • The `ulong` type represents unsigned 64-bit integers with values between 0 and
9 18446744073709551615 .
- 10 • The `char` type represents unsigned 16-bit integers with values between 0 and 65535. The set of possible
11 values for the `char` type corresponds to the Unicode character set. [*Note*: Although `char` has the same
12 representation as `ushort`, not all operations permitted on one type are permitted on the other. *end note*]

13 The integral-type unary and binary operators always operate with signed 32-bit precision, unsigned 32-bit
14 precision, signed 64-bit precision, or unsigned 64-bit precision:

- 15 • For the unary `+` and `~` operators, the operand is converted to type `T`, where `T` is the first of `int`, `uint`,
16 `long`, and `ulong` that can fully represent all possible values of the operand. The operation is then
17 performed using the precision of type `T`, and the type of the result is `T`.
- 18 • For the unary `-` operator, the operand is converted to type `T`, where `T` is the first of `int` and `long` that
19 can fully represent all possible values of the operand. The operation is then performed using the
20 precision of type `T`, and the type of the result is `T`. The unary `-` operator cannot be applied to operands of
21 type `ulong`.
- 22 • For the binary `+`, `-`, `*`, `/`, `%`, `&`, `^`, `|`, `==`, `!=`, `>`, `<`, `>=`, and `<=` operators, the operands are converted to
23 type `T`, where `T` is the first of `int`, `uint`, `long`, and `ulong` that can fully represent all possible values of
24 both operands. The operation is then performed using the precision of type `T`, and the type of the result is
25 `T` (or `bool` for the relational operators). It is not permitted for one operand to be of type `long` and the
26 other to be of type `ulong` with the binary operators.
- 27 • For the binary `<<` and `>>` operators, the left operand is converted to type `T`, where `T` is the first of `int`,
28 `uint`, `long`, and `ulong` that can fully represent all possible values of the operand. The operation is then
29 performed using the precision of type `T`, and the type of the result is `T`.

30 The `char` type is classified as an integral type, but it differs from the other integral types in two ways:

- 31 • There are no implicit conversions from other types to the `char` type. In particular, even though the
32 `sbyte`, `byte`, and `ushort` types have ranges of values that are fully representable using the `char` type,
33 implicit conversions from `sbyte`, `byte`, or `ushort` to `char` do not exist.
- 34 • Constants of the `char` type must be written as *character-literals* or as *integer-literals* in combination
35 with a cast to type `char`. For example, `(char)10` is the same as `'\x000A'`.

36 The `checked` and `unchecked` operators and statements are used to control overflow checking for integral-
37 type arithmetic operations and conversions (§14.5.12). In a `checked` context, an overflow produces a
38 compile-time error or causes a `System.OverflowException` to be thrown. In an `unchecked` context,
39 overflows are ignored and any high-order bits that do not fit in the destination type are discarded.

40 11.1.5 Floating point types

41 C# supports two floating-point types: `float` and `double`. The `float` and `double` types are represented
42 using the 32-bit single-precision and 64-bit double-precision IEEE 754 formats, which provide the following
43 sets of values:

- 1 • Positive zero and negative zero. In most situations, positive zero and negative zero behave identically as
2 the simple value zero, but certain operations distinguish between the two (§14.7.2).
- 3 • Positive infinity and negative infinity. Infinities are produced by such operations as dividing a non-zero
4 number by zero. For example, `1.0 / 0.0` yields positive infinity, and `-1.0 / 0.0` yields negative
5 infinity.
- 6 • The *Not-a-Number* value, often abbreviated NaN. NaNs are produced by invalid floating-point
7 operations, such as dividing zero by zero.
- 8 • The finite set of non-zero values of the form $s \times m \times 2^e$, where s is 1 or -1 , and m and e are determined
9 by the particular floating-point type: For `float`, $0 < m < 2^{24}$ and $-149 \leq e \leq 104$, and for `double`,
10 $0 < m < 2^{53}$ and $-1075 \leq e \leq 970$. Denormalized floating-point numbers are considered valid non-zero
11 values.

12 The `float` type can represent values ranging from approximately 1.5×10^{-45} to 3.4×10^{38} with a precision
13 of 7 digits.

14 The `double` type can represent values ranging from approximately 5.0×10^{-324} to 1.7×10^{308} with a
15 precision of 15–16 digits.

16 If one of the operands of a binary operator is of a floating-point type, then the other operand must be of an
17 integral type or a floating-point type, and the operation is evaluated as follows:

- 18 • If one of the operands is of an integral type, then that operand is converted to the floating-point type of
19 the other operand.
- 20 • Then, if either of the operands is of type `double`, the other operand is converted to `double`, the
21 operation is performed using at least `double` range and precision, and the type of the result is `double`
22 (or `bool` for the relational operators).
- 23 • Otherwise, the operation is performed using at least `float` range and precision, and the type of the
24 result is `float` (or `bool` for the relational operators).

25 The floating-point operators, including the assignment operators, never produce exceptions. Instead, in
26 exceptional situations, floating-point operations produce zero, infinity, or NaN, as described below:

- 27 • If the result of a floating-point operation is too small for the destination format, the result of the
28 operation becomes positive zero or negative zero.
- 29 • If the result of a floating-point operation is too large for the destination format, the result of the
30 operation becomes positive infinity or negative infinity.
- 31 • If a floating-point operation is invalid, the result of the operation becomes NaN.
- 32 • If one or both operands of a floating-point operation is NaN, the result of the operation becomes NaN.

33 Floating-point operations may be performed with higher precision than the result type of the operation. For
34 example, some hardware architectures support an “extended” or “long double” floating-point type with
35 greater range and precision than the `double` type, and implicitly perform all floating-point operations using
36 this higher precision type. Only at excessive cost in performance can such hardware architectures be made to
37 perform floating-point operations with *less* precision, and rather than require an implementation to forfeit
38 both performance and precision, C# allows a higher precision type to be used for all floating-point
39 operations. Other than delivering more precise results, this rarely has any measurable effects. However, in
40 expressions of the form `x * y / z`, where the multiplication produces a result that is outside the `double`
41 range, but the subsequent division brings the temporary result back into the `double` range, the fact that the
42 expression is evaluated in a higher range format may cause a finite result to be produced instead of an
43 infinity.

11.1.6 The decimal type

The `decimal` type is a 128-bit data type suitable for financial and monetary calculations. The `decimal` type can represent values ranging from 1.0×10^{-28} to approximately 7.9×10^{28} with 28–29 significant digits.

The finite set of values of type `decimal` are of the form $-1^s \times c \times 10^e$, where the sign s is 0 or 1, the coefficient c is given by $0 \leq c < 2^{96}$, and the scale e is such that $0 \leq e \leq 28$. The `decimal` type does not support signed zeros, infinities, or NaN's.

A `decimal` is represented as a 96-bit integer scaled by a power of ten. For `decimals` with an absolute value less than 1.0m, the value is exact to the 28th decimal place, but no further. For `decimals` with an absolute value greater than or equal to 1.0m, the value is exact to 28 or 29 digits. Contrary to the `float` and `double` data types, decimal fractional numbers such as 0.1 can be represented exactly in the `decimal` representation. In the `float` and `double` representations, such numbers are often infinite fractions, making those representations more prone to round-off errors.

If one of the operands of a binary operator is of type `decimal`, then the other operand must be of an integral type or of type `decimal`. If an integral type operand is present, it is converted to `decimal` before the operation is performed.

The result of an operation on values of type `decimal` is that which would result from calculating an exact result (preserving scale, as defined for each operator) and then rounding to fit the representation. Results are rounded to the nearest representable value, and, when a result is equally close to two representable values, to the value that has an even number in the least significant digit position (this is known as “banker’s rounding”). That is, results are exact to 28 or 29 digits, but to no more than 28 decimal places. A zero result always has a sign of 0 and a scale of 0.

If a decimal arithmetic operation produces a value that is too small for the decimal format after rounding, the result of the operation becomes zero. If a `decimal` arithmetic operation produces a result that is too large for the `decimal` format, a `System.OverflowException` is thrown.

The `decimal` type has greater precision but smaller range than the floating-point types. Thus, conversions from the floating-point types to `decimal` might produce overflow exceptions, and conversions from `decimal` to the floating-point types might cause loss of precision. For these reasons, no implicit conversions exist between the floating-point types and `decimal`, and without explicit casts, it is not possible to mix floating-point and `decimal` operands in the same expression.

11.1.7 The bool type

The `bool` type represents boolean logical quantities. The possible values of type `bool` are `true` and `false`.

No standard conversions exist between `bool` and other types. In particular, the `bool` type is distinct and separate from the integral types, and a `bool` value cannot be used in place of an integral value, and vice versa.

[*Note:* In the C and C++ languages, a zero integral or floating-point value, or a null pointer can be converted to the boolean value `false`, and a non-zero integral or floating-point value, or a non-null pointer can be converted to the boolean value `true`. In C#, such conversions are accomplished by explicitly comparing an integral or floating-point value to zero, or by explicitly comparing an object reference to `null`. *end note*]

11.1.8 Enumeration types

An enumeration type is a distinct type with named constants. Every enumeration type has an underlying type, which must be `byte`, `sbyte`, `short`, `ushort`, `int`, `uint`, `long` or `ulong`. Enumeration types are defined through enumeration declarations (§21.1).

11.2 Reference types

A reference type is a class type, an interface type, an array type, or a delegate type.

```

1      reference-type:
2          class-type
3          interface-type
4          array-type
5          delegate-type
6
7      class-type:
8          type-name
9          object
10         string
11
12     interface-type:
13         type-name
14
15     array-type:
16         non-array-type rank-specifiers
17
18     non-array-type:
19         type
20
21     rank-specifiers:
22         rank-specifier
23         rank-specifiers rank-specifier
24
25     rank-specifier:
26         [ dim-separatorsopt ]
27
28     dim-separators:
29         ,
30         dim-separators ,
31
32     delegate-type:
33         type-name

```

26 A reference type value is a reference to an *instance* of the type, the latter known as an *object*. The special
 27 value `null` is compatible with all reference types and indicates the absence of an instance.

28 11.2.1 Class types

29 A class type defines a data structure that contains data members (constants and fields), function members
 30 (methods, properties, events, indexers, operators, instance constructors, destructors, and static constructors),
 31 and nested types. Class types support inheritance, a mechanism whereby derived classes can extend and
 32 specialize base classes. Instances of class types are created using *object-creation-expressions* (§14.5.10.1).

33 Class types are described in §17.

34 11.2.2 The object type

35 The `object` class type is the ultimate base class of all other types. Every type in C# directly or indirectly
 36 derives from the `object` class type.

37 The keyword `object` is simply an alias for the predefined class `System.Object`.

38 11.2.3 The string type

39 The `string` type is a sealed class type that inherits directly from `object`. Instances of the `string` class
 40 represent Unicode character strings.

41 Values of the `string` type can be written as string literals (§9.4.4).

42 The keyword `string` is simply an alias for the predefined class `System.String`.

1 11.2.4 Interface types

2 An interface defines a contract. A class or struct that implements an interface must adhere to its contract. An
3 interface may inherit from multiple base interfaces, and a class or struct may implement multiple interfaces.

4 Interface types are described in §20.

5 11.2.5 Array types

6 An array is a data structure that contains zero or more variables which are accessed through computed
7 indices. The variables contained in an array, also called the elements of the array, are all of the same type,
8 and this type is called the element type of the array.

9 Array types are described in §19.

10 11.2.6 Delegate types

11 A delegate is a data structure that refers to one or more methods, and for instance methods, it also refers to
12 their corresponding object instances.

13 [*Note:* The closest equivalent of a delegate in C or C++ is a function pointer, but whereas a function pointer
14 can only reference static functions, a delegate can reference both static and instance methods. In the latter
15 case, the delegate stores not only a reference to the method's entry point, but also a reference to the object
16 instance on which to invoke the method. *end note*]

17 Delegate types are described in §22.

18 11.3 Boxing and unboxing

19 The concept of boxing and unboxing is central to C#'s type system. It provides a bridge between *value-types*
20 and *reference-types* by permitting any value of a *value-type* to be converted to and from type **object**.

21 Boxing and unboxing enables a unified view of the type system wherein a value of any type can ultimately
22 be treated as an object.

23 11.3.1 Boxing conversions

24 A boxing conversion permits any *value-type* to be implicitly converted to the type **object** or to any
25 *interface-type* implemented by the *value-type*. Boxing a value of a *value-type* consists of allocating an object
26 instance and copying the *value-type* value into that instance.

27 The actual process of boxing a value of a *value-type* is best explained by imagining the existence of a **boxing**
28 **class** for that type. [*Example:* For any *value-type* T, the boxing class behaves as if it were declared as
29 follows:

```
30     sealed class T_Box
31     {
32         T value;
33         public T_Box(T t) {
34             value = t;
35         }
36     }
```

37 Boxing of a value *v* of type T now consists of executing the expression `new T_Box(v)`, and returning the
38 resulting instance as a value of type **object**. Thus, the statements

```
39     int i = 123;
40     object box = i;
```

41 conceptually correspond to

```
42     int i = 123;
43     object box = new int_Box(i);
```

44 *end example*]

1 Boxing classes like `T_Box` and `int_Box` above don't actually exist and the dynamic type of a boxed value
 2 isn't actually a class type. Instead, a boxed value of type `T` has the dynamic type `T`, and a dynamic type
 3 check using the `is` operator can simply reference type `T`. [Example: For example,

```

4     int i = 123;
5     object box = i;
6     if (box is int) {
7         Console.WriteLine("Box contains an int");
8     }

```

9 will output the string “Box contains an int” on the console. *end example*]

10 A boxing conversion implies *making a copy* of the value being boxed. This is different from a conversion of
 11 a *reference-type* to type `object`, in which the value continues to reference the same instance and simply is
 12 regarded as the less derived type `object`. [Example: For example, given the declaration

```

13     struct Point
14     {
15         public int x, y;
16         public Point(int x, int y) {
17             this.x = x;
18             this.y = y;
19         }
20     }

```

21 the following statements

```

22     Point p = new Point(10, 10);
23     object box = p;
24     p.x = 20;
25     Console.WriteLine(((Point)box).x);

```

26 will output the value 10 on the console because the implicit boxing operation that occurs in the assignment
 27 of `p` to `box` causes the value of `p` to be copied. Had `Point` been declared a `class` instead, the value 20
 28 would be output because `p` and `box` would reference the same instance. *end example*]

29 11.3.2 Unboxing conversions

30 An unboxing conversion permits an explicit conversion from type `object` to any *value-type* or from any
 31 *interface-type* to any *value-type* that implements the *interface-type*. An unboxing operation consists of first
 32 checking that the object instance is a boxed value of the given *value-type*, and then copying the value out of
 33 the instance.

34 Referring to the imaginary boxing class described in the previous section, an unboxing conversion of an
 35 object `box` to a *value-type* `T` consists of executing the expression `((T_Box)box).value`. [Example: Thus,
 36 the statements

```

37     object box = 123;
38     int i = (int)box;

```

39 conceptually correspond to

```

40     object box = new int_Box(123);
41     int i = ((int_Box)box).value;

```

42 *end example*]

43 For an unboxing conversion to a given *value-type* to succeed at run-time, the value of the source operand
 44 must be a reference to an object that was previously created by boxing a value of that *value-type*. If the
 45 source operand is `null` or a reference to an incompatible object, a `System.InvalidCastException` is
 46 thrown.

12. Variables

1

2 Variables represent storage locations. Every variable has a type that determines what values can be stored in
 3 the variable. C# is a type-safe language, and the C# compiler guarantees that values stored in variables are
 4 always of the appropriate type. The value of a variable can be changed through assignment or through use of
 5 the ++ and -- operators.

6 A variable must be *definitely assigned* (§12.3) before its value can be obtained.

7 As described in the following sections, variables are either *initially assigned* or *initially unassigned*. An
 8 initially assigned variable has a well-defined initial value and is always considered definitely assigned. An
 9 initially unassigned variable has no initial value. For an initially unassigned variable to be considered
 10 definitely assigned at a certain location, an assignment to the variable must occur in every possible execution
 11 path leading to that location.

12.1 Variable categories

13 C# defines seven categories of variables: static variables, instance variables, array elements, value
 14 parameters, reference parameters, output parameters, and local variables. The sections that follow describe
 15 each of these categories.

16 [*Example:* In the example

```

17     class A
18     {
19         public static int x;
20         int y;
21
22         void F(int[] v, int a, ref int b, out int c) {
23             int i = 1;
24             c = a + b++;
25         }
26     }
  
```

26 x is a static variable, y is an instance variable, v[0] is an array element, a is a value parameter, b is a
 27 reference parameter, c is an output parameter, and i is a local variable. *end example*]

12.1.1 Static variables

29 A field declared with the `static` modifier is called a *static variable*. A static variable comes into existence
 30 before execution of the static constructor (§17.11) for its containing type, and ceases to exist when the
 31 associated application domain ceases to exist..

32 The initial value of a static variable is the default value (§12.2) of the variable's type.

33 For the purposes of definite assignment checking, a static variable is considered initially assigned.

12.1.2 Instance variables

35 A field declared without the `static` modifier is called an *instance variable*.

12.1.2.1 Instance variables in classes

37 An instance variable of a class comes into existence when a new instance of that class is created, and ceases
 38 to exist when there are no references to that instance and the instance's destructor (if any) has executed.

39 The initial value of an instance variable of a class is the default value (§12.2) of the variable's type.

40 For the purpose of definite assignment checking, an instance variable is considered initially assigned.

1 12.1.2.2 Instance variables in structs

2 An instance variable of a struct has exactly the same lifetime as the struct variable to which it belongs. In
3 other words, when a variable of a struct type comes into existence or ceases to exist, so too do the instance
4 variables of the struct.

5 The initial assignment state of an instance variable of a struct is the same as that of the containing struct
6 variable. In other words, when a struct variable is considered initially assigned, so too are its instance
7 variables, and when a struct variable is considered initially unassigned, its instance variables are likewise
8 unassigned.

9 **12.1.3 Array elements**

10 The elements of an array come into existence when an array instance is created, and cease to exist when
11 there are no references to that array instance.

12 The initial value of each of the elements of an array is the default value (§12.2) of the type of the array
13 elements.

14 For the purpose of definite assignment checking, an array element is considered initially assigned.

15 **12.1.4 Value parameters**

16 A parameter declared without a `ref` or `out` modifier is a *value parameter*.

17 A value parameter comes into existence upon invocation of the function member (method, instance
18 constructor, accessor, or operator) to which the parameter belongs, and is initialized with the value of the
19 argument given in the invocation. A value parameter ceases to exist upon return of the function member.

20 For the purpose of definite assignment checking, a value parameter is considered initially assigned.

21 **12.1.5 Reference parameters**

22 A parameter declared with a `ref` modifier is a *reference parameter*.

23 A reference parameter does not create a new storage location. Instead, a reference parameter represents the
24 same storage location as the variable given as the argument in the function member invocation. Thus, the
25 value of a reference parameter is always the same as the underlying variable.

26 The following definite assignment rules apply to reference parameters. Note the different rules for output
27 parameters described in §12.1.6.

28 • A variable must be definitely assigned (§12.3) before it can be passed as a reference parameter in a
29 function member invocation.

30 • Within a function member, a reference parameter is considered initially assigned.

31 Within an instance method or instance accessor of a struct type, the `this` keyword behaves exactly as a
32 reference parameter of the struct type (§14.5.7).

33 **12.1.6 Output parameters**

34 A parameter declared with an `out` modifier is an *output parameter*.

35 An output parameter does not create a new storage location. Instead, an output parameter represents the
36 same storage location as the variable given as the argument in the function member invocation. Thus, the
37 value of an output parameter is always the same as the underlying variable.

38 The following definite assignment rules apply to output parameters. Note the different rules for reference
39 parameters described in §12.1.5.

40 • A variable need not be definitely assigned before it can be passed as an output parameter in a function
41 member invocation.

- 1 • Following the normal completion of a function member invocation, each variable that was passed as an
2 output parameter is considered assigned in that execution path.
- 3 • Within a function member, an output parameter is considered initially unassigned.
- 4 • Every output parameter of a function member must be definitely assigned (§12.3) before the function
5 member returns normally.

6 Within an instance constructor of a struct type, the `this` keyword behaves exactly as an output parameter of
7 the struct type (§14.5.7).

8 12.1.7 Local variables

9 A *local variable* is declared by a *local-variable-declaration*, which may occur in a *block*, a *for-statement*, a
10 *switch-statement*, or a *using-statement*.

11 The lifetime of a local variable is the portion of program execution during which storage is guaranteed to be
12 reserved for it. This lifetime extends from entry into the *block*, *for-statement*, *switch-statement*, or *using-*
13 *statement* with which it is associated, until execution of that *block*, *for-statement*, *switch-statement*, or *using-*
14 *statement* ends in any way. (Entering an enclosed *block* or calling a method suspends, but does not end,
15 execution of the current *block*, *for-statement*, *switch-statement*, or *using-statement*.) If the parent *block*, *for-*
16 *statement*, *switch-statement*, or *using-statement* is entered recursively, a new instance of the local variable is
17 created each time, and its *local-variable-initializer*, if any, is evaluated each time.

18 A local variable is not automatically initialized and thus has no default value. For the purpose of definite
19 assignment checking, a local variable is considered initially unassigned. A *local-variable-declaration* may
20 include a *local-variable-initializer*, in which case the variable is considered definitely assigned in its entire
21 scope, except within the expression provided in the *local-variable-initializer*.

22 Within the scope of a local variable, it is a compile-time error to refer to that local variable in a textual
23 position that precedes its *local-variable-declarator*.

24 [Note: The actual lifetime of a local variable is implementation-dependent. For example, a compiler might
25 statically determine that a local variable in a block is only used for a small portion of that block. Using this
26 analysis, the compiler could generate code that results in the variable's storage having a shorter lifetime than
27 its containing block.

28 The storage referred to by a local reference variable is reclaimed independently of the lifetime of that local
29 reference variable (§10.9). *end note*]

30 A local variable is also declared by a *foreach-statement* and by a *specific-catch-clause* for a *try-statement*.
31 For a *foreach-statement*, the local variable is an iteration variable (§15.8.4). For a *specific-catch-clause*, the
32 local variable is an exception variable (§15.10). A local variable declared by a *foreach-statement* or *specific-*
33 *catch-clause* is considered definitely assigned in its entire scope.

34 12.2 Default values

35 The following categories of variables are automatically initialized to their default values:

- 36 • Static variables.
- 37 • Instance variables of class instances.
- 38 • Array elements.

39 The default value of a variable depends on the type of the variable and is determined as follows:

- 40 • For a variable of a *value-type*, the default value is the same as the value computed by the *value-type*'s
41 default constructor (§11.1.1).
- 42 • For a variable of a *reference-type*, the default value is `null`.

1 [Note: Initialization to default values is typically done by having the memory manager or garbage collector
2 initialize memory to all-bits-zero before it is allocated for use. For this reason, it is convenient to use all-bits-
3 zero to represent the null reference. *end note*]

4 **12.3 Definite assignment**

5 At a given location in the executable code of a function member, a variable is said to be *definitely assigned*
6 if the compiler can prove, by static flow analysis, that the variable has been automatically initialized or has
7 been the target of at least one assignment. The rules of definite assignment are:

- 8 • An initially assigned variable (§12.3.1) is always considered definitely assigned.
- 9 • An initially unassigned variable (§12.3.2) is considered definitely assigned at a given location if all
10 possible execution paths leading to that location contain at least one of the following:
 - 11 ○ A simple assignment (§14.13.1) in which the variable is the left operand.
 - 12 ○ An invocation expression (§14.5.5) or object creation expression (§14.5.10.1) that passes the
13 variable as an output parameter.
 - 14 ○ For a local variable, a local variable declaration (§15.5) that includes a variable initializer.

15 The definite assignment states of instance variables of a *struct-type* variable are tracked individually as well
16 as collectively. In addition to the rules above, the following rules apply to *struct-type* variables and their
17 instance variables:

- 18 • An instance variable is considered definitely assigned if its containing *struct-type* variable is considered
19 definitely assigned.
- 20 • A *struct-type* variable is considered definitely assigned if each of its instance variables is considered
21 definitely assigned.

22 Definite assignment is a requirement in the following contexts:

- 23 • A variable must be definitely assigned at each location where its value is obtained. [Note: This ensures
24 that undefined values never occur. *end note*] The occurrence of a variable in an expression is considered
25 to obtain the value of the variable, except when
 - 26 ○ the variable is the left operand of a simple assignment,
 - 27 ○ the variable is passed as an output parameter, or
 - 28 ○ the variable is a *struct-type* variable and occurs as the left operand of a member access.
- 29 • A variable must be definitely assigned at each location where it is passed as a reference parameter.
30 [Note: This ensures that the function member being invoked can consider the reference parameter
31 initially assigned. *end note*]
- 32 • All output parameters of a function member must be definitely assigned at each location where the
33 function member returns (through a `return` statement or through execution reaching the end of the
34 function member body). [Note: This ensures that function members do not return undefined values in
35 output parameters, thus enabling the compiler to consider a function member invocation that takes a
36 variable as an output parameter equivalent to an assignment to the variable. *end note*]
- 37 • The `this` variable of a *struct-type* instance constructor must be definitely assigned at each location
38 where that instance constructor returns.

39 **12.3.1 Initially assigned variables**

40 The following categories of variables are classified as initially assigned:

- 41 • Static variables.
- 42 • Instance variables of class instances.

- 1 • Instance variables of initially assigned struct variables.
- 2 • Array elements.
- 3 • Value parameters.
- 4 • Reference parameters.
- 5 • Variables declared in a `catch` clause or a `foreach` statement.

6 **12.3.2 Initially unassigned variables**

7 The following categories of variables are classified as initially unassigned:

- 8 • Instance variables of initially unassigned struct variables.
- 9 • Output parameters, including the `this` variable of struct instance constructors.
- 10 • Local variables, except those declared in a `catch` clause or a `foreach` statement.

11 **12.3.3 Precise rules for determining definite assignment**

12 In order to determine that each used variable is definitely assigned, the compiler must use a process that is
13 equivalent to the one described in this section.

14 The compiler processes the body of each function member that has one or more initially unassigned
15 variables. For each initially unassigned variable v , the compiler determines a *definite assignment state* for v
16 at each of the following points in the function member:

- 17 • At the beginning of each statement
- 18 • At the end point (§15.1) of each statement
- 19 • On each arc which transfers control to another statement or to the end point of a statement
- 20 • At the beginning of each expression
- 21 • At the end of each expression

22 The definite assignment state of v can be either:

- 23 • Definitely assigned. This indicates that on all possible control flows to this point, v has been
24 assigned a value.
- 25 • Not definitely assigned. For the state of a variable at the end of an expression of type `bool`, the state
26 of a variable the isn't definitely assigned may (but doesn't necessarily) fall into one of the following
27 sub-states:
 - 28 ○ Definitely assigned after true expression. This state indicates that v is definitely assigned if
29 the boolean expression evaluated as true, but is not necessarily assigned if the boolean
30 expression evaluated as false.
 - 31 ○ Definitely assigned after false expression. This state indicates that v is definitely assigned if
32 the boolean expression evaluated as false, but is not necessarily assigned if the boolean
33 expression evaluated as true.

34 The following rules govern how the state of a variable v is determined at each location.

35 **12.3.3.1 General rules for statements**

- 36 • v is not definitely assigned at the beginning of a function member body.
- 37 • v is definitely assigned at the beginning of any unreachable statement.
- 38 • The definite assignment state of v at the beginning of any other statement is determined by checking
39 the definite assignment state of v on all control flow transfers that target the beginning of that

1 statement. If (and only if) v is definitely assigned on all such control flow transfers, then v is
 2 definitely assigned at the beginning of the statement. The set of possible control flow transfers is
 3 determined in the same way as for checking statement reachability (§15.1).

- 4 • The definite assignment state of v at the end point of a block, `checked`, `unchecked`, `if`, `while`,
 5 `do`, `for`, `foreach`, `lock`, `using`, or `switch` statement is determined by checking the definite
 6 assignment state of v on all control flow transfers that target the end point of that statement. If v is
 7 definitely assigned on all such control flow transfers, then v is definitely assigned at the end point of
 8 the statement. Otherwise, v is not definitely assigned at the end point of the statement. The set of
 9 possible control flow transfers is determined in the same way as for checking statement reachability
 10 (§15.1).

11 12.3.3.2 Block statements, checked, and unchecked statements

12 The definite assignment state of v on the control transfer to the first statement of the statement list in the
 13 block (or to the end point of the block, if the statement list is empty) is the same as the definite assignment
 14 statement of v before the block, `checked`, or `unchecked` statement.

15 12.3.3.3 Expression statements

16 For an expression statement *stmt* that consists of the expression *expr*:

- 17 • v has the same definite assignment state at the beginning of *expr* as at the beginning of *stmt*.
- 18 • If v is definitely assigned at the end of *expr*, it is definitely assigned at the end point of *stmt*;
 19 otherwise; it is not definitely assigned at the end point of *stmt*.

20 12.3.3.4 Declaration statements

- 21 • If *stmt* is a declaration statement without initializers, then v has the same definite assignment state at
 22 the end point of *stmt* as at the beginning of *stmt*.
- 23 • If *stmt* is a declaration statement with initializers, then the definite assignment state for v is
 24 determined as if *stmt* were a statement list, with one assignment statement for each declaration with
 25 an initializer (in the order of declaration).

26 12.3.3.5 If statements

27 For an `if` statement *stmt* of the form:

28 `if (expr) then-stmt else else-stmt`

- 29 • v has the same definite assignment state at the beginning of *expr* as at the beginning of *stmt*.
- 30 • If v is definitely assigned at the end of *expr*, then it is definitely assigned on the control flow transfer
 31 to *then-stmt* and to either *else-stmt* or to the end-point of *stmt* if there is no else clause.
- 32 • If v has the state “definitely assigned after true expression” at the end of *expr*, then it is definitely
 33 assigned on the control flow transfer to *then-stmt*, and not definitely assigned on the control flow
 34 transfer to either *else-stmt* or to the end-point of *stmt* if there is no else clause.
- 35 • If v has the state “definitely assigned after false expression” at the end of *expr*, then it is definitely
 36 assigned on the control flow transfer to *else-stmt*, and not definitely assigned on the control flow
 37 transfer to *then-stmt*. It is definitely assigned at the end-point of *stmt* if and only if it is definitely
 38 assigned at the end-point of *then-stmt*.
- 39 • Otherwise, v is considered not definitely assigned on the control flow transfer to either the *then-stmt*
 40 or *else-stmt*, or to the end-point of *stmt* if there is no else clause.

41 12.3.3.6 Switch statements

42 In a `switch` statement *stmt* with controlling expression *expr*:

- 1 • The definite assignment state of v at the beginning of $expr$ is the same as the state of v at the
- 2 beginning of $stmt$.
- 3 • The definite assignment state of v on the control flow transfer to a reachable switch block statement
- 4 list is the same as the definite assignment state of v at the end of $expr$.

5 12.3.3.7 While statements

6 For a `while` statement $stmt$ of the form:

```
7     while ( $expr$ )  $while-body$ 
```

- 8 • v has the same definite assignment state at the beginning of $expr$ as at the beginning of $stmt$.
- 9 • If v is definitely assigned at the end of $expr$, then it is definitely assigned on the control flow transfer
- 10 to $while-body$ and to the end point of $stmt$.
- 11 • If v has the state “definitely assigned after true expression” at the end of $expr$, then it is definitely
- 12 assigned on the control flow transfer to $while-body$, but not definitely assigned at the end-point of
- 13 $stmt$.
- 14 • If v has the state “definitely assigned after false expression” at the end of $expr$, then it is definitely
- 15 assigned on the control flow transfer to the end point of $stmt$.

16 12.3.3.8 Do statements

17 For a `do` statement $stmt$ of the form:

```
18     do  $do-body$  while ( $expr$ );
```

- 19 • v has the same definite assignment state on the control flow transfer from the beginning of $stmt$ to
- 20 $do-body$ as at the beginning of $stmt$.
- 21 • v has the same definite assignment state at the beginning of $expr$ as at the end point of $do-body$.
- 22 • If v is definitely assigned at the end of $expr$, then it is definitely assigned on the end point of $stmt$.
- 23 • If v has the state “definitely assigned after false expression” at the end of $expr$, then it is definitely
- 24 assigned on the control flow transfer to the end point of $stmt$.

25 12.3.3.9 For statements

26 Definite assignment checking for a `for` statement of the form:

```
27     for ( $for-initializer$ ;  $for-condition$ ;  $for-iterator$ )  $embedded-statement$ 
```

28 is done as if the statement were written:

```
29     {
30          $for-initializer$ ;
31         while ( $for-condition$ ) {
32              $embedded-statement$ ;
33              $for-iterator$ ;
34         }
35     }
```

36 If the $for-condition$ is omitted from the `for` statement, then evaluation of definite assignment proceeds as if

37 $for-condition$ were replaced with `true` in the above expansion.

38 12.3.3.10 Break, continue, and goto statements

39 The definite assignment state of v on the control flow transfer caused by a `break`, `continue`, or `goto`

40 statement is the same as the definite assignment state of v at the beginning of the statement.

1 12.3.3.11 Throw statements

2 For a statement *stmt* of the form

3 `throw expr ;`

4 The definite assignment state of *v* at the beginning of *expr* is the same as the definite assignment state of *v* at
5 the beginning of *stmt*.

6 12.3.3.12 Return statements

7 For a statement *stmt* of the form

8 `return expr ;`

- 9 • The definite assignment state of *v* at the beginning of *expr* is the same as the definite assignment
10 state of *v* at the beginning of *stmt*.
- 11 • If *v* is an output parameter, then it must be definitely assigned either:
- 12 ○ after *expr*
 - 13 ○ or at the end of the `finally` block of a `try-finally` or `try-catch-finally` that
14 encloses the `return` statement.

15 12.3.3.13 Try-catch statements

16 For a statement *stmt* of the form:

17 `try try-block`
18 `catch(...) catch-block-1`
19 `...`
20 `catch(...) catch-block-n`
21

- 22 • The definite assignment state of *v* at the beginning of *try-block* is the same as the definite
23 assignment state of *v* at the beginning of *stmt*.
- 24 • The definite assignment state of *v* at the beginning of *catch-block-i* (for any *i*) is the same as the
25 definite assignment state of *v* at the beginning of *stmt*.
- 26 • The definite assignment state of *v* at the end-point of *stmt* is definitely assigned if (and only if) *v* is
27 definitely assigned at the end-point of *try-block* and every *catch-block-i* (for every *i* from 1 to *n*).

28 12.3.3.14 Try-finally statements

29 For a `try` statement *stmt* of the form:

30 `try try-block finally finally-block`

- 31 • The definite assignment state of *v* at the beginning of *try-block* is the same as the definite
32 assignment state of *v* at the beginning of *stmt*.
- 33 • The definite assignment state of *v* at the beginning of *finally-block* is the same as the definite
34 assignment state of *v* at the beginning of *stmt*.
- 35 • The definite assignment state of *v* at the end-point of *stmt* is definitely assigned if (and only if)
36 either:
- 37 ○ *v* is definitely assigned at the end-point of *try-block*
 - 38 ○ *v* is definitely assigned at the end-point of *finally-block*

39 If a control flow transfer (for example, a `goto` statement) is made that begins within *try-block*, and ends
40 outside of *try-block*, then *v* is also considered definitely assigned on that control flow transfer if *v* is

1 definitely assigned at the end-point of *finally-block*. (This is not an only if—if *v* is definitely assigned for
2 another reason on this control flow transfer, then it is still considered definitely assigned.)

3 12.3.3.15 Try-catch-finally statements

4 Definite assignment analysis for a `try-catch-finally` statement of the form:

```
5     try try-block
6     catch(...) catch-block-1
7     ...
8     catch(...) catch-block-n
9     finally finally-block
```

10 is done as if the statement were a `try-finally` statement enclosing a `try-catch` statement:

```
11     try {
12         try try-block
13         catch(...) catch-block-1
14         ...
15         catch(...) catch-block-n
16     }
17     finally finally-block
```

18 [Example: The following example demonstrates how the different blocks of a `try` statement (§15.10) affect
19 definite assignment.

```
20     class A
21     {
22     static void F() {
23         int i, j;
24         try {
25             goto LABEL:
26             // neither i nor j definitely assigned
27             i = 1;
28             // i definitely assigned
29         }
30         catch {
31             // neither i nor j definitely assigned
32             i = 3;
33             // i definitely assigned
34         }
35         finally {
36             // neither i nor j definitely assigned
37             j = 5;
38             // j definitely assigned
39         }
40         // i and j definitely assigned
41     LABEL:
42         // j definitely assigned
43     }
44 }
45 }
```

46 *end example*]

47 12.3.3.16 Foreach statements

48 For a `foreach` statement *stmt* of the form:

```
49     foreach (type identifier in expr) embedded-statement
```

- 50 • The definite assignment state of *v* at the beginning of *expr* is the same as the state of *v* at the
51 beginning of *stmt*.
- 52 • The definite assignment state of *v* on the control flow transfer to *embedded-statement* or to the end
53 point of *stmt* is the same as the state of *v* at the end of *expr*.

1 12.3.3.17 Using statements

2 For a `using` statement *stmt* of the form:

3 `using (resource-acquisition) embedded-statement`

- 4 • The definite assignment state of *v* at the beginning of *resource-acquisition* is the same as the state of
5 *v* at the beginning of *stmt*.
- 6 • The definite assignment state of *v* on the control flow transfer to *embedded-statement* is the same as
7 the state of *v* at the end of *resource-acquisition*.

8 12.3.3.18 Lock statements

9 For a `lock` statement *stmt* of the form:

10 `lock (expr) embedded-statement`

- 11 • The definite assignment state of *v* at the beginning of *expr* is the same as the state of *v* at the
12 beginning of *stmt*.
- 13 • The definite assignment state of *v* on the control flow transfer to *embedded-statement* is the same as
14 the state of *v* at the end of *expr*.

15 12.3.3.19 General rules for simple expressions

16 The following rule applies to these kinds of expressions: literals (§14.5.1), simple names (§14.5.2), member
17 access expressions (§14.5.4), non-indexed base access expressions (§14.5.8), and `typeof` expressions
18 (§14.5.11).

- 19 • The definite assignment state of *v* at the end of such an expression is the same as the definite
20 assignment state of *v* at the beginning of the expression.

21 12.3.3.20 General rules for expressions with embedded expressions

22 The following rules apply to these kinds of expressions: parenthesized expressions (§14.5.3), element access
23 expressions (§14.5.6), base access expressions with indexing (§14.5.8), increment and decrement
24 expressions (§14.5.9, §14.6.5), cast expressions (§14.6.6), unary `+`, `-`, `~`, `*` expressions, binary `+`, `-`, `*`, `/`, `%`,
25 `<<`, `>>`, `<`, `<=`, `>`, `>=`, `==`, `!=`, `is`, `as`, `&`, `|`, `^` expressions (§14.7, §14.8, §14.9, §14.10), compound
26 assignment expressions (§14.13.2), `checked` and `unchecked` expressions (§14.5.12), array and delegate
27 creation expressions (§14.5.10).

28 Each of these expressions has one or more sub-expressions that are unconditionally evaluated in a fixed
29 order. For example, the binary `%` operator evaluates the left hand side of the operator, then the right hand
30 side. An indexing operation evaluates the indexed expression, and then evaluates each of the index
31 expressions, in order from left to right. For an expression *expr*, which has sub-expressions *expr*₁, *expr*₂, ...,
32 *expr*_{*n*}, evaluated in that order:

- 33 • The definite assignment state of *v* at the beginning of *expr*_{*i*} is the same as the definite assignment
34 state at the beginning of *expr*.
- 35 • The definite assignment state of *v* at the beginning of *expr*_{*i*} (*i* greater than one) is the same as the
36 definite assignment state at the end of *expr*_{*i-1*}.
- 37 • The definite assignment state of *v* at the end of *expr* is the same as the definite assignment state at
38 the end of *expr*_{*n*}.

39 12.3.3.21 Invocation expressions and object creation expressions

40 For an invocation expression *expr* of the form:

41 `primary-expression (arg1, arg2, ..., argn)`

42 or an object creation expression of the form:

1 `new type (arg1, arg2, ..., argn)`

- 2 • For an invocation expression, the definite assignment state of v before *primary-expression* is the
3 same as the state of v before *expr*.
- 4 • For an invocation expression, the definite assignment state of v before arg_i is the same as the state of
5 v after *primary-expression*.
- 6 • For an object creation expression, the definite assignment state of v before arg_i is the same as the
7 state of v before *expr*.
- 8 • For each argument arg_i , the definite assignment state of v after arg_i is determined by the normal
9 expression rules, ignoring any `ref` or `out` modifiers.
- 10 • For each argument arg_i for any i greater than one, the definite assignment state of v before arg_i is the
11 same as the state of v after arg_{i-1} .
- 12 • If the variable v is passed as an `out` argument (i.e., an argument of the form “`out v`”) in any of the
13 arguments, then the state of v after *expr* is definitely assigned. Otherwise; the state of v after *expr* is
14 the same as the state of v after arg_n .

15 12.3.3.22 Simple assignment expressions

16 For an expression *expr* of the form $w = expr\text{-rhs}$:

- 17 • The definite assignment state of v before *expr-rhs* is the same as the definite assignment state of v
18 before *expr*.
- 19 • If w is the same variable as v , then the definite assignment state of v after *expr* is definitely assigned.
20 Otherwise, the definite assignment state of v after *expr* is the same as the definite assignment state of
21 v after *expr-rhs*.

22 12.3.3.23 && expressions

23 For an expression *expr* of the form *expr-first* && *expr-second*:

- 24 • The definite assignment state of v before *expr-first* is the same as the definite assignment state of v
25 before *expr*.
- 26 • The definite assignment state of v before *expr-second* is definitely assigned if the state of v after
27 *expr-first* is either definitely assigned or “definitely assigned after true expression”. Otherwise, it is
28 not definitely assigned.
- 29 • The definite assignment statement of v after *expr* is determined by:
 - 30 ○ If the state of v after *expr-first* is definitely assigned, then the state of v after *expr* is
31 definitely assigned.
 - 32 ○ Otherwise, if the state of v after *expr-second* is definitely assigned, and the state of v after
33 *expr-first* is “definitely assigned after false expression”, then the state of v after *expr* is
34 definitely assigned.
 - 35 ○ Otherwise, if the state of v after *expr-second* is definitely assigned or “definitely assigned
36 after true expression”, then the state of v after *expr* is “definitely assigned after true
37 expression”.
 - 38 ○ Otherwise, if the state of v after *expr-first* is “definitely assigned after false expression”, and
39 the state of v after *expr-second* is “definitely assigned after false expression”, then the state
40 of v after *expr* is “definitely assigned after false expression”.
 - 41 ○ Otherwise, the state of v after *expr* is not definitely assigned.

42 [Example: In the example

```

1      class A
2      {
3          static void F(int x, int y) {
4              int i;
5              if (x >= 0 && (i = y) >= 0) {
6                  // i definitely assigned
7              }
8              else {
9                  // i not definitely assigned
10             }
11             // i not definitely assigned
12         }
13     }

```

14 the variable `i` is considered definitely assigned in one of the embedded statements of an `if` statement but not
15 in the other. In the `if` statement in method `F`, the variable `i` is definitely assigned in the first embedded
16 statement because execution of the expression `(i = y)` always precedes execution of this embedded
17 statement. In contrast, the variable `i` is not definitely assigned in the second embedded statement, since
18 `x >= 0` might have tested false, resulting in the variable `i`'s being unassigned. *end example*]

19 12.3.3.24 `||` expressions

20 For an expression `expr` of the form `expr-first || expr-second`:

- 21 • The definite assignment state of `v` before `expr-first` is the same as the definite assignment state of `v`
22 before `expr`.
- 23 • The definite assignment state of `v` before `expr-second` is definitely assigned if the state of `v` after
24 `expr-first` is either definitely assigned or “definitely assigned after false expression”. Otherwise, it is
25 not definitely assigned.
- 26 • The definite assignment statement of `v` after `expr` is determined by:
 - 27 ○ If the state of `v` after `expr-first` is definitely assigned, then the state of `v` after `expr` is
28 definitely assigned.
 - 29 ○ Otherwise, if the state of `v` after `expr-second` is definitely assigned, and the state of `v` after
30 `expr-first` is “definitely assigned after true expression”, then the state of `v` after `expr` is
31 definitely assigned.
 - 32 ○ Otherwise, if the state of `v` after `expr-second` is definitely assigned or “definitely assigned
33 after false expression”, then the state of `v` after `expr` is “definitely assigned after false
34 expression”.
 - 35 ○ Otherwise, if the state of `v` after `expr-first` is “definitely assigned after true expression”, and
36 the state of `v` after `expr-second` is “definitely assigned after true expression”, then the state
37 of `v` after `expr` is “definitely assigned after true expression”.
 - 38 ○ Otherwise, the state of `v` after `expr` is not definitely assigned.

39 [*Example:* In the example

```

40     class A
41     {
42         static void G(int x, int y) {
43             int i;
44             if (x >= 0 || (i = y) >= 0) {
45                 // i not definitely assigned
46             }
47             else {
48                 // i definitely assigned
49             }
50             // i not definitely assigned
51         }
52     }

```

1 the variable `i` is considered definitely assigned in one of the embedded statements of an `if` statement but not
 2 in the other. In the `if` statement in method `G`, the variable `i` is definitely assigned in the second embedded
 3 statement because execution of the expression `(i = y)` always precedes execution of this embedded
 4 statement. In contrast, the variable `i` is not definitely assigned in the first embedded statement, since
 5 `x >= 0` might have tested false, resulting in the variable `i`'s being unassigned. *end example*]

6 12.3.3.25 ! expressions

7 For an expression `expr` of the form `! expr-operand`:

- 8 • The definite assignment state of `v` before `expr-operand` is the same as the definite assignment state
 9 of `v` before `expr`.
- 10 • The definite assignment state of `v` after `expr` is determined by:
 - 11 ○ If the state of `v` after `expr-operand` is definitely assigned, then the state of `v` after `expr` is
 12 definitely assigned.
 - 13 ○ If the state of `v` after `expr-operand` is not definitely assigned, then the state of `v` after `expr` is
 14 not definitely assigned.
 - 15 ○ If the state of `v` after `expr-operand` is “definitely assigned after false expression”, then the
 16 state of `v` after `expr` is “definitely assigned after true expression”.
 - 17 ○ If the state of `v` after `expr-operand` is “definitely assigned after true expression”, then the
 18 state of `v` after `expr` is “definitely assigned after false expression”.

19 12.3.3.26 ?: expressions

20 For an expression `expr` of the form `expr-cond ? expr-true : expr-false`:

- 21 • The definite assignment state of `v` before `expr-cond` is the same as the state of `v` before `expr`.
- 22 • The definite assignment state of `v` before `expr-true` is definitely assigned if and only if the state of `v`
 23 after `expr-cond` is definitely assigned or “definitely assigned after true expression”.
- 24 • The definite assignment state of `v` before `expr-false` is definitely assigned if and only if the state of `v`
 25 after `expr-cond` is definitely assigned or “definitely assigned after false expression”.

26 12.4 Variable references

27 A *variable-reference* is an *expression* that is classified as a variable. A *variable-reference* denotes a storage
 28 location that can be accessed both to fetch the current value and to store a new value.

29 *variable-reference*:
 30 *expression*

31 [Note: In C and C++, a *variable-reference* is known as an *lvalue*. *end note*]

32 12.5 Atomicity of variable references

33 Reads and writes of the following data types shall be atomic: `bool`, `char`, `byte`, `sbyte`, `short`, `ushort`,
 34 `uint`, `int`, `float`, and reference types. In addition, reads and writes of enum types with an underlying type
 35 in the previous list shall also be atomic. Reads and writes of other types, including `long`, `ulong`, `double`,
 36 and `decimal`, as well as user-defined types, need not be atomic. Aside from the library functions designed
 37 for that purpose, there is no guarantee of atomic read-modify-write, such as in the case of increment or
 38 decrement.

39

13. Conversions

1

2 A *conversion* enables an expression of one type to be treated as another type. Conversions can be *implicit* or
 3 *explicit*, and this determines whether an explicit cast is required. [*Example*: For instance, the conversion
 4 from type `int` to type `long` is implicit, so expressions of type `int` can implicitly be treated as type `long`.
 5 The opposite conversion, from type `long` to type `int`, is explicit and so an explicit cast is required.

```
6     int a = 123;
7     long b = a;      // implicit conversion from int to long
8     int c = (int) b; // explicit conversion from long to int
```

9 *end example*] Some conversions are defined by the language. Programs may also define their own
 10 conversions (§13.4).

11 13.1 Implicit conversions

12 The following conversions are classified as implicit conversions:

- 13 • Identity conversions
- 14 • Implicit numeric conversions
- 15 • Implicit enumeration conversions.
- 16 • Implicit reference conversions
- 17 • Boxing conversions
- 18 • Implicit constant expression conversions
- 19 • User-defined implicit conversions

20 Implicit conversions can occur in a variety of situations, including function member invocations (§14.4.3),
 21 cast expressions (§14.6.6), and assignments (§14.13).

22 The pre-defined implicit conversions always succeed and never cause exceptions to be thrown. [*Note*:
 23 Properly designed user-defined implicit conversions should exhibit these characteristics as well. *end note*]

24 13.1.1 Identity conversion

25 An identity conversion converts from any type to the same type. This conversion exists only such that an
 26 entity that already has a required type can be said to be convertible to that type.

27 13.1.2 Implicit numeric conversions

28 The implicit numeric conversions are:

- 29 • From `sbyte` to `short`, `int`, `long`, `float`, `double`, or `decimal`.
- 30 • From `byte` to `short`, `ushort`, `int`, `uint`, `long`, `ulong`, `float`, `double`, or `decimal`.
- 31 • From `short` to `int`, `long`, `float`, `double`, or `decimal`.
- 32 • From `ushort` to `int`, `uint`, `long`, `ulong`, `float`, `double`, or `decimal`.
- 33 • From `int` to `long`, `float`, `double`, or `decimal`.
- 34 • From `uint` to `long`, `ulong`, `float`, `double`, or `decimal`.
- 35 • From `long` to `float`, `double`, or `decimal`.

- 1 • From `ulong` to `float`, `double`, or `decimal`.
 - 2 • From `char` to `ushort`, `int`, `uint`, `long`, `ulong`, `float`, `double`, or `decimal`.
 - 3 • From `float` to `double`.
- 4 Conversions from `int`, `uint`, or `long` to `float` and from `long` to `double` may cause a loss of precision,
5 but will never cause a loss of magnitude. The other implicit numeric conversions never lose any information.
6 There are no implicit conversions to the `char` type, so values of the other integral types do not automatically
7 convert to the `char` type.

8 **13.1.3 Implicit enumeration conversions**

9 An implicit enumeration conversion permits the *decimal-integer-literal* `0` to be converted to any *enum-type*.

10 **13.1.4 Implicit reference conversions**

11 The implicit reference conversions are:

- 12 • From any *reference-type* to `object`.
- 13 • From any *class-type* `S` to any *class-type* `T`, provided `S` is derived from `T`.
- 14 • From any *class-type* `S` to any *interface-type* `T`, provided `S` implements `T`.
- 15 • From any *interface-type* `S` to any *interface-type* `T`, provided `S` is derived from `T`.
- 16 • From an *array-type* `S` with an element type `SE` to an *array-type* `T` with an element type `TE`, provided all
17 of the following are true:
 - 18 ○ `S` and `T` differ only in element type. In other words, `S` and `T` have the same number of dimensions.
 - 19 ○ Both `SE` and `TE` are *reference-types*.
 - 20 ○ An implicit reference conversion exists from `SE` to `TE`.
- 21 • From any *array-type* to `System.Array`.
- 22 • From any *delegate-type* to `System.Delegate`.
- 23 • From any *array-type* or *delegate-type* to `System.ICloneable`.
- 24 • From the null type to any *reference-type*.

25 The implicit reference conversions are those conversions between *reference-types* that can be proven to
26 always succeed, and therefore require no checks at run-time.

27 Reference conversions, implicit or explicit, never change the referential identity of the object being
28 converted. [Note: In other words, while a reference conversion may change the type of the reference, it never
29 changes the type or value of the object being referred to. *end note*]

30 **13.1.5 Boxing conversions**

31 A boxing conversion permits any *value-type* to be implicitly converted to the type `object` or to any
32 *interface-type* implemented by the *value-type*. Boxing a value of a *value-type* consists of allocating an object
33 instance and copying the *value-type* value into that instance.

34 Boxing conversions are described further in §11.3.1.

35 **13.1.6 Implicit constant expression conversions**

36 An implicit constant expression conversion permits the following conversions:

- 37 • A *constant-expression* (§14.15) of type `int` can be converted to type `sbyte`, `byte`, `short`, `ushort`,
38 `uint`, or `ulong`, provided the value of the *constant-expression* is within the range of the destination
39 type.

- 1 • A *constant-expression* of type `long` can be converted to type `ulong`, provided the value of the *constant-*
2 *expression* is not negative.

3 13.1.7 User-defined implicit conversions

4 A user-defined implicit conversion consists of an optional standard implicit conversion, followed by
5 execution of a user-defined implicit conversion operator, followed by another optional standard implicit
6 conversion. The exact rules for evaluating user-defined conversions are described in §13.4.3.

7 13.2 Explicit conversions

8 The following conversions are classified as explicit conversions:

- 9 • All implicit conversions.
- 10 • Explicit numeric conversions.
- 11 • Explicit enumeration conversions.
- 12 • Explicit reference conversions.
- 13 • Explicit interface conversions.
- 14 • Unboxing conversions.
- 15 • User-defined explicit conversions.

16 Explicit conversions can occur in cast expressions (§14.6.6).

17 The set of explicit conversions includes all implicit conversions. [*Note:* This means that redundant cast
18 expressions are allowed. *end note*]

19 The explicit conversions that are not implicit conversions are conversions that cannot be proven to always
20 succeed, conversions that are known to possibly lose information, and conversions across domains of types
21 sufficiently different to merit explicit notation.

22 13.2.1 Explicit numeric conversions

23 The explicit numeric conversions are the conversions from a *numeric-type* to another *numeric-type* for
24 which an implicit numeric conversion (§13.1.2) does not already exist:

- 25 • From `sbyte` to `byte`, `ushort`, `uint`, `ulong`, or `char`.
- 26 • From `byte` to `sbyte` and `char`.
- 27 • From `short` to `sbyte`, `byte`, `ushort`, `uint`, `ulong`, or `char`.
- 28 • From `ushort` to `sbyte`, `byte`, `short`, or `char`.
- 29 • From `int` to `sbyte`, `byte`, `short`, `ushort`, `uint`, `ulong`, or `char`.
- 30 • From `uint` to `sbyte`, `byte`, `short`, `ushort`, `int`, or `char`.
- 31 • From `long` to `sbyte`, `byte`, `short`, `ushort`, `int`, `uint`, `ulong`, or `char`.
- 32 • From `ulong` to `sbyte`, `byte`, `short`, `ushort`, `int`, `uint`, `long`, or `char`.
- 33 • From `char` to `sbyte`, `byte`, or `short`.
- 34 • From `float` to `sbyte`, `byte`, `short`, `ushort`, `int`, `uint`, `long`, `ulong`, `char`, or `decimal`.
- 35 • From `double` to `sbyte`, `byte`, `short`, `ushort`, `int`, `uint`, `long`, `ulong`, `char`, `float`, or `decimal`.
- 36 • From `decimal` to `sbyte`, `byte`, `short`, `ushort`, `int`, `uint`, `long`, `ulong`, `char`, `float`, or `double`.

37 Because the explicit conversions include all implicit and explicit numeric conversions, it is always possible
38 to convert from any *numeric-type* to any other *numeric-type* using a cast expression (§14.6.6).

1 The explicit numeric conversions possibly lose information or possibly cause exceptions to be thrown. An
2 explicit numeric conversion is processed as follows:

- 3 • For a conversion from an integral type to another integral type, the processing depends on the overflow
4 checking context (§14.5.12) in which the conversion takes place:
 - 5 ○ In a **checked** context, the conversion succeeds if the value of the source operand is within the range
6 of the destination type, but throws a `System.OverflowException` if the value of the source
7 operand is outside the range of the destination type.
 - 8 ○ In an **unchecked** context, the conversion always succeeds, and proceeds as follows.
 - 9 • If the source type is larger than the destination type, then the source value is truncated by
10 discarding its “extra” most significant bits. The result is then treated as a value of the destination
11 type.
 - 12 • If the source type is smaller than the destination type, then the source value is either sign-
13 extended or zero-extended so that it is the same size as the destination type. Sign-extension is
14 used if the source type is signed; zero-extension is used if the source type is unsigned. The result
15 is then treated as a value of the destination type.
 - 16 • If the source type is the same size as the destination type, then the source value is treated as a
17 value of the destination type
- 18 • For a conversion from `decimal` to an integral type, the source value is rounded towards zero to the
19 nearest integral value, and this integral value becomes the result of the conversion. If the resulting
20 integral value is outside the range of the destination type, a `System.OverflowException` is thrown.
- 21 • For a conversion from `float` or `double` to an integral type, the processing depends on the overflow-
22 checking context (§14.5.12) in which the conversion takes place:
 - 23 ○ In a **checked** context, the conversion proceeds as follows:
 - 24 • If the value of the source operand is within the range of the destination type, then it is rounded
25 towards zero to the nearest integral value of the destination type, and this integral value is the
26 result of the conversion.
 - 27 • Otherwise, a `System.OverflowException` is thrown.
 - 28 ○ In an **unchecked** context, the conversion always succeeds, and proceeds as follows.
 - 29 • If the value of the source operand is within the range of the destination type, then it is rounded
30 towards zero to the nearest integral value of the destination type, and this integral value is the
31 result of the conversion.
 - 32 • Otherwise, the result of the conversion is an unspecified value of the destination type.
- 33 • For a conversion from `double` to `float`, the `double` value is rounded to the nearest `float` value. If
34 the `double` value is too small to represent as a `float`, the result becomes positive zero or negative zero.
35 If the `double` value is too large to represent as a `float`, the result becomes positive infinity or negative
36 infinity. If the `double` value is NaN, the result is also NaN.
- 37 • For a conversion from `float` or `double` to `decimal`, the source value is converted to `decimal`
38 representation and rounded to the nearest number after the 28th decimal place if required (§11.1.6). If the
39 source value is too small to represent as a `decimal`, the result becomes zero. If the source value is NaN,
40 infinity, or too large to represent as a `decimal`, a `System.OverflowException` is thrown.
- 41 • For a conversion from `decimal` to `float` or `double`, the `decimal` value is rounded to the nearest
42 `double` or `float` value. While this conversion may lose precision, it never causes an exception to be
43 thrown.

1 13.2.2 Explicit enumeration conversions

2 The explicit enumeration conversions are:

- 3 • From `sbyte`, `byte`, `short`, `ushort`, `int`, `uint`, `long`, `ulong`, `char`, `float`, `double`, or `decimal` to
4 any *enum-type*.
- 5 • From any *enum-type* to `sbyte`, `byte`, `short`, `ushort`, `int`, `uint`, `long`, `ulong`, `char`, `float`,
6 `double`, or `decimal`.
- 7 • From any *enum-type* to any other *enum-type*.

8 An explicit enumeration conversion between two types is processed by treating any participating *enum-type*
9 as the underlying type of that *enum-type*, and then performing an implicit or explicit numeric conversion
10 between the resulting types. For example, given an *enum-type* `E` with an underlying type of `int`, a
11 conversion from `E` to `byte` is processed as an explicit numeric conversion (§13.2.1) from `int` to `byte`, and
12 a conversion from `byte` to `E` is processed as an implicit numeric conversion (§13.1.2) from `byte` to `int`.

13 13.2.3 Explicit reference conversions

14 The explicit reference conversions are:

- 15 • From `object` to any *reference-type*.
- 16 • From any *class-type* `S` to any *class-type* `T`, provided `S` is a base class of `T`.
- 17 • From any *class-type* `S` to any *interface-type* `T`, provided `S` is not sealed and provided `S` does not
18 implement `T`.
- 19 • From any *interface-type* `S` to any *class-type* `T`, provided `T` is not sealed or provided `T` implements `S`.
- 20 • From any *interface-type* `S` to any *interface-type* `T`, provided `S` is not derived from `T`.
- 21 • From an *array-type* `S` with an element type `SE` to an *array-type* `T` with an element type `TE`, provided all
22 of the following are true:
 - 23 ○ `S` and `T` differ only in element type. (In other words, `S` and `T` have the same number of dimensions.)
 - 24 ○ Both `SE` and `TE` are *reference-types*.
 - 25 ○ An explicit reference conversion exists from `SE` to `TE`.
- 26 • From `System.Array` and the interfaces it implements, to any *array-type*.
- 27 • From `System.Delegate` and the interfaces it implements, to any *delegate-type*.

28 The explicit reference conversions are those conversions between reference-types that require run-time
29 checks to ensure they are correct.

30 For an explicit reference conversion to succeed at run-time, the value of the source operand must be `null`,
31 or the *actual* type of the object referenced by the source operand must be a type that can be converted to the
32 destination type by an implicit reference conversion (§13.1.4). If an explicit reference conversion fails, a
33 `System.InvalidCastException` is thrown.

34 Reference conversions, implicit or explicit, never change the referential identity of the object being
35 converted. [Note: In other words, while a reference conversion may change the type of the reference, it never
36 changes the type or value of the object being referred to. *end note*]

37 13.2.4 Unboxing conversions

38 An unboxing conversion permits an explicit conversion from type `object` to any *value-type* or from any
39 *interface-type* to any *value-type* that implements the *interface-type*. An unboxing operation consists of first
40 checking that the object instance is a boxed value of the given *value-type*, and then copying the value out of
41 the instance.

42 Unboxing conversions are described further in §11.3.2.

1 **13.2.5 User-defined explicit conversions**

2 A user-defined explicit conversion consists of an optional standard explicit conversion, followed by
3 execution of a user-defined implicit or explicit conversion operator, followed by another optional standard
4 explicit conversion. The exact rules for evaluating user-defined conversions are described in §13.4.4.

5 **13.3 Standard conversions**

6 The standard conversions are those pre-defined conversions that can occur as part of a user-defined
7 conversion.

8 **13.3.1 Standard implicit conversions**

9 The following implicit conversions are classified as standard implicit conversions:

- 10 • Identity conversions (§13.1.1)
- 11 • Implicit numeric conversions (§13.1.2)
- 12 • Implicit reference conversions (§13.1.4)
- 13 • Boxing conversions (§13.1.5)
- 14 • Implicit constant expression conversions (§13.1.6)

15 The standard implicit conversions specifically exclude user-defined implicit conversions.

16 **13.3.2 Standard explicit conversions**

17 The standard explicit conversions are all standard implicit conversions plus the subset of the explicit
18 conversions for which an opposite standard implicit conversion exists. [*Note:* In other words, if a standard
19 implicit conversion exists from a type A to a type B, then a standard explicit conversion exists from type A to
20 type B and from type B to type A. *end note*]

21 **13.4 User-defined conversions**

22 C# allows the pre-defined implicit and explicit conversions to be augmented by *user-defined conversions*.
23 User-defined conversions are introduced by declaring conversion operators (§17.9.3) in class and struct
24 types.

25 **13.4.1 Permitted user-defined conversions**

26 C# permits only certain user-defined conversions to be declared. In particular, it is not possible to redefine
27 an already existing implicit or explicit conversion. A class or struct is permitted to declare a conversion from
28 a source type S to a target type T only if all of the following are true:

- 29 • S and T are different types.
- 30 • Either S or T is the class or struct type in which the operator declaration takes place.
- 31 • Neither S nor T is `object` or an *interface-type*.
- 32 • T is not a base class of S, and S is not a base class of T.

33 The restrictions that apply to user-defined conversions are discussed further in §17.9.3.

34 **13.4.2 Evaluation of user-defined conversions**

35 A user-defined conversion converts a value from its type, called the *source type*, to another type, called the
36 *target type*. Evaluation of a user-defined conversion centers on finding the *most specific* user-defined
37 conversion operator for the particular source and target types. This determination is broken into several
38 steps:

- 1 • Finding the set of classes and structs from which user-defined conversion operators will be considered.
2 This set consists of the source type and its base classes and the target type and its base classes (with the
3 implicit assumptions that only classes and structs can declare user-defined operators, and that non-class
4 types have no base classes).
- 5 • From that set of types, determining which user-defined conversion operators are applicable. For a
6 conversion operator to be applicable, it must be possible to perform a standard conversion (§13.3) from
7 the source type to the operand type of the operator, and it must be possible to perform a standard
8 conversion from the result type of the operator to the target type.
- 9 • From the set of applicable user-defined operators, determining which operator is unambiguously the
10 most specific. In general terms, the most specific operator is the operator whose operand type is
11 “closest” to the source type and whose result type is “closest” to the target type. The exact rules for
12 establishing the most specific user-defined conversion operator are defined in the following sections.

13 Once a most specific user-defined conversion operator has been identified, the actual execution of the user-
14 defined conversion involves up to three steps:

- 15 • First, if required, performing a standard conversion from the source type to the operand type of the user-
16 defined conversion operator.
- 17 • Next, invoking the user-defined conversion operator to perform the conversion.
- 18 • Finally, if required, performing a standard conversion from the result type of the user-defined
19 conversion operator to the target type.

20 Evaluation of a user-defined conversion never involves more than one user-defined conversion operator. In
21 other words, a conversion from type S to type T will never first execute a user-defined conversion from S to
22 X and then execute a user-defined conversion from X to T.

23 Exact definitions of evaluation of user-defined implicit or explicit conversions are given in the following
24 sections. The definitions make use of the following terms:

- 25 • If a standard implicit conversion (§13.3.1) exists from a type A to a type B, and if neither A nor B are
26 *interface-types*, then A is said to be **encompassed by** B, and B is said to **encompass** A.
- 27 • The **most encompassing type** in a set of types is the one type that encompasses all other types in the set.
28 If no single type encompasses all other types, then the set has no most encompassing type. In more
29 intuitive terms, the most encompassing type is the “largest” type in the set—the one type to which each
30 of the other types can be implicitly converted.
- 31 • The **most encompassed type** in a set of types is the one type that is encompassed by all other types in the
32 set. If no single type is encompassed by all other types, then the set has no most encompassed type. In
33 more intuitive terms, the most encompassed type is the “smallest” type in the set—the one type that can
34 be implicitly converted to each of the other types.

35 13.4.3 User-defined implicit conversions

36 A user-defined implicit conversion from type S to type T is processed as follows:

- 37 • Find the set of types, D, from which user-defined conversion operators will be considered. This set
38 consists of S (if S is a class or struct), the base classes of S (if S is a class), T (if T is a class or struct),
39 and the base classes of T (if T is a class).
- 40 • Find the set of applicable user-defined conversion operators, U. This set consists of the user-defined
41 implicit conversion operators declared by the classes or structs in D that convert from a type
42 encompassing S to a type encompassed by T. If U is empty, the conversion is undefined and a compile-
43 time error occurs.
- 44 • Find the most specific source type, S_x , of the operators in U:
45 ○ If any of the operators in U convert from S, then S_x is S.

- 1 ○ Otherwise, S_x is the most encompassed type in the combined set of source types of the operators
2 in U . If no most encompassed type can be found, then the conversion is ambiguous and a compile-
3 time error occurs.
- 4 • Find the most specific target type, T_x , of the operators in U :
 - 5 ○ If any of the operators in U convert to T , then T_x is T .
 - 6 ○ Otherwise, T_x is the most encompassing type in the combined set of target types of the operators
7 in U . If no most encompassing type can be found, then the conversion is ambiguous and a compile-
8 time error occurs.
- 9 • If U contains exactly one user-defined conversion operator that converts from S_x to T_x , then this is the
10 most specific conversion operator. If no such operator exists, or if more than one such operator exists,
11 then the conversion is ambiguous and a compile-time error occurs. Otherwise, the user-defined
12 conversion is applied:
 - 13 ○ If S is not S_x , then a standard implicit conversion from S to S_x is performed.
 - 14 ○ The most specific user-defined conversion operator is invoked to convert from S_x to T_x .
 - 15 ○ If T_x is not T , then a standard implicit conversion from T_x to T is performed.

16 **13.4.4 User-defined explicit conversions**

17 A user-defined explicit conversion from type S to type T is processed as follows:

- 18 • Find the set of types, D , from which user-defined conversion operators will be considered. This set
19 consists of S (if S is a class or struct), the base classes of S (if S is a class), T (if T is a class or struct),
20 and the base classes of T (if T is a class).
- 21 • Find the set of applicable user-defined conversion operators, U . This set consists of the user-defined
22 implicit or explicit conversion operators declared by the classes or structs in D that convert from a type
23 encompassing or encompassed by S to a type encompassing or encompassed by T . If U is empty, the
24 conversion is undefined and a compile-time error occurs.
- 25 • Find the most specific source type, S_x , of the operators in U :
 - 26 ○ If any of the operators in U convert from S , then S_x is S .
 - 27 ○ Otherwise, if any of the operators in U convert from types that encompass S , then S_x is the most
28 encompassed type in the combined set of source types of those operators. If no most encompassed
29 type can be found, then the conversion is ambiguous and a compile-time error occurs.
 - 30 ○ Otherwise, S_x is the most encompassing type in the combined set of source types of the operators
31 in U . If no most encompassing type can be found, then the conversion is ambiguous and a compile-
32 time error occurs.
- 33 • Find the most specific target type, T_x , of the operators in U :
 - 34 ○ If any of the operators in U convert to T , then T_x is T .
 - 35 ○ Otherwise, if any of the operators in U convert to types that are encompassed by T , then T_x is the
36 most encompassing type in the combined set of source types of those operators. If no most
37 encompassing type can be found, then the conversion is ambiguous and a compile-time error occurs.
 - 38 ○ Otherwise, T_x is the most encompassed type in the combined set of target types of the operators in U .
39 If no most encompassed type can be found, then the conversion is ambiguous and a compile-time
40 error occurs.
- 41 • If U contains exactly one user-defined conversion operator that converts from S_x to T_x , then this is the
42 most specific conversion operator. If no such operator exists, or if more than one such operator exists,
43 then the conversion is ambiguous and a compile-time error occurs. Otherwise, the user-defined
44 conversion is applied:

- 1 ○ If S is not S_x , then a standard explicit conversion from S to S_x is performed.
- 2 ○ The most specific user-defined conversion operator is invoked to convert from S_x to T_x .
- 3 ○ If T_x is not T , then a standard explicit conversion from T_x to T is performed.

14. Expressions

1

2 An expression is a sequence of operators and operands. This chapter defines the syntax, order of evaluation
3 of operands and operators, and meaning of expressions.

4 14.1 Expression classifications

5 An expression is classified as one of the following:

- 6 • A value. Every value has an associated type.
- 7 • A variable. Every variable has an associated type, namely the declared type of the variable.
- 8 • A namespace. An expression with this classification can only appear as the left-hand side of a *member-*
9 *access* (§14.5.4). In any other context, an expression classified as a namespace causes a compile-time
10 error.
- 11 • A type. An expression with this classification can only appear as the left-hand side of a *member-access*
12 (§14.5.4), or as an operand for the `as` operator (§14.9.10), the `is` operator (§14.9.9), or the `typeof`
13 operator (§14.5.11). In any other context, an expression classified as a type causes a compile-time error.
- 14 • A method group, which is a set of overloaded methods resulting from a member lookup (§14.3). A
15 method group may have an associated instance expression. When an instance method is invoked, the
16 result of evaluating the instance expression becomes the instance represented by `this` (§14.5.7). A
17 method group is only permitted in an *invocation-expression* (§14.5.5) or a *delegate-creation-expression*
18 (§14.5.10.3). In any other context, an expression classified as a method group causes a compile-time
19 error.
- 20 • A property access. Every property access has an associated type, namely the type of the property.
21 Furthermore, a property access may have an associated instance expression. When an accessor (the `get`
22 or `set` block) of an instance property access is invoked, the result of evaluating the instance expression
23 becomes the instance represented by `this` (§14.5.7).
- 24 • An event access. Every event access has an associated type, namely the type of the event. Furthermore,
25 an event access may have an associated instance expression. An event access may appear as the left-
26 hand operand of the `+=` and `-=` operators (§14.13.3). In any other context, an expression classified as an
27 event access causes a compile-time error.
- 28 • An indexer access. Every indexer access has an associated type, namely the element type of the
29 indexer. Furthermore, an indexer access has an associated instance expression and an associated
30 argument list. When an accessor (the `get` or `set` block) of an indexer access is invoked, the result of
31 evaluating the instance expression becomes the instance represented by `this` (§14.5.7), and the result of
32 evaluating the argument list becomes the parameter list of the invocation.
- 33 • Nothing. This occurs when the expression is an invocation of a method with a return type of `void`. An
34 expression classified as nothing is only valid in the context of a *statement-expression* (§15.6).

35 The final result of an expression is never a namespace, type, method group, or event access. Rather, as noted
36 above, these categories of expressions are intermediate constructs that are only permitted in certain contexts.

37 A property access or indexer access is always reclassified as a value by performing an invocation of the *get-*
38 *accessor* or the *set-accessor*. The particular accessor is determined by the context of the property or indexer
39 access: If the access is the target of an assignment, the *set-accessor* is invoked to assign a new value
40 (§14.13.1). Otherwise, the *get-accessor* is invoked to obtain the current value (§14.1.1).

14.1.1 Values of expressions

Most of the constructs that involve an expression ultimately require the expression to denote a *value*. In such cases, if the actual expression denotes a namespace, a type, a method group, or nothing, a compile-time error occurs. However, if the expression denotes a property access, an indexer access, or a variable, the value of the property, indexer, or variable is implicitly substituted:

- The value of a variable is simply the value currently stored in the storage location identified by the variable. A variable must be considered definitely assigned (§12.3) before its value can be obtained, or otherwise a compile-time error occurs.
- The value of a property access expression is obtained by invoking the *get-accessor* of the property. If the property has no *get-accessor*, a compile-time error occurs. Otherwise, a function member invocation (§14.4.3) is performed, and the result of the invocation becomes the value of the property access expression.
- The value of an indexer access expression is obtained by invoking the *get-accessor* of the indexer. If the indexer has no *get-accessor*, a compile-time error occurs. Otherwise, a function member invocation (§14.4.3) is performed with the argument list associated with the indexer access expression, and the result of the invocation becomes the value of the indexer access expression.

14.2 Operators

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

There are three kinds of operators:

- Unary operators. The unary operators take one operand and use either prefix notation (such as -x) or postfix notation (such as x++).
- Binary operators. The binary operators take two operands and all use infix notation (such as x + y).
- Ternary operator. Only one ternary operator, ?:, exists; it takes three operands and uses infix notation (c ? x : y).

The order of evaluation of operators in an expression is determined by the *precedence* and *associativity* of the operators (§14.2.1).

The order in which operands in an expression are evaluated, is left to right. [*Example:* For example, in $F(i) + G(i++) * H(i)$, method F is called using the old value of i, then method G is called with the old value of i, and, finally, method H is called with the new value of i. This is separate from and unrelated to operator precedence. *end example*] Certain operators can be *overloaded*. Operator overloading permits user-defined operator implementations to be specified for operations where one or both of the operands are of a user-defined class or struct type (§14.2.2).

14.2.1 Operator precedence and associativity

When an expression contains multiple operators, the *precedence* of the operators controls the order in which the individual operators are evaluated. [*Note:* For example, the expression $x + y * z$ is evaluated as $x + (y * z)$ because the * operator has higher precedence than the binary + operator. *end note*] The precedence of an operator is established by the definition of its associated grammar production. [*Note:* For example, an *additive-expression* consists of a sequence of *multiplicative-expressions* separated by + or - operators, thus giving the + and - operators lower precedence than the *, /, and % operators. *end note*]

The following table summarizes all operators in order of precedence from highest to lowest:

Section	Category	Operators
14.5	Primary	x.y f(x) a[x] x++ x-- new typeof checked unchecked
14.6	Unary	+ - ! ~ ++x --x (T)x
14.7	Multiplicative	* / %
14.7	Additive	+ -
14.8	Shift	<< >>
14.9	Relational and type-testing	< > <= >= is as
14.9	Equality	== !=
14.10	Logical AND	&
14.10	Logical XOR	^
14.10	Logical OR	
14.11	Conditional AND	&&
14.11	Conditional OR	
14.12	Conditional	?:
14.13	Assignment	= *= /= %= += -= <<= >>= &= ^= =

1

2

When an operand occurs between two operators with the same precedence, the *associativity* of the operators controls the order in which the operations are performed:

3

- 4 • Except for the assignment operators, all binary operators are *left-associative*, meaning that operations
5 are performed from left to right. [Example: For example, $x + y + z$ is evaluated as $(x + y) + z$.
6 *end example*]
- 7 • The assignment operators and the conditional operator (`?:`) are *right-associative*, meaning that
8 operations are performed from right to left. [Example: For example, $x = y = z$ is evaluated as
9 $x = (y = z)$. *end example*]

10

11

12

Precedence and associativity can be controlled using parentheses. [Example: For example, $x + y * z$ first
multiplies y by z and then adds the result to x , but $(x + y) * z$ first adds x and y and then multiplies the
result by z . *end example*]

13

14.2.2 Operator overloading

14

15

16

17

18

All unary and binary operators have predefined implementations that are automatically available in any
expression. In addition to the predefined implementations, user-defined implementations can be introduced
by including `operator` declarations in classes and structs (§17.9). User-defined operator implementations
always take precedence over predefined operator implementations: Only when no applicable user-defined
operator implementations exist will the predefined operator implementations be considered.

19

The *overloadable unary operators* are:

20

+ - ! ~ ++ -- true false

21

22

23

[Note: Although `true` and `false` are not used explicitly in expressions, they are considered operators
because they are invoked in several expression contexts: boolean expressions (§14.16) and expressions
involving the conditional (§14.12), and conditional logical operators (§14.11). *end note*]

24

The *overloadable binary operators* are:

25

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

1 Only the operators listed above can be overloaded. In particular, it is not possible to overload member
 2 access, method invocation, or the =, &&, ||, ?:, checked, unchecked, new, typeof, as, and
 3 is operators.

4 When a binary operator is overloaded, the corresponding assignment operator, if any, is also implicitly
 5 overloaded. For example, an overload of operator * is also an overload of operator *=. This is described
 6 further in §14.13. Note that the assignment operator itself (=) cannot be overloaded. An assignment always
 7 performs a simple bit-wise copy of a value into a variable.

8 Cast operations, such as (T)x, are overloaded by providing user-defined conversions (§13.4).

9 Element access, such as a[x], is not considered an overloadable operator. Instead, user-defined indexing is
 10 supported through indexers (§17.8).

11 In expressions, operators are referenced using operator notation, and in declarations, operators are referenced
 12 using functional notation. The following table shows the relationship between operator and functional
 13 notations for unary and binary operators. In the first entry, *op* denotes any overloadable unary prefix
 14 operator. In the second entry, *op* denotes the unary postfix ++ and -- operators. In the third entry, *op*
 15 denotes any overloadable binary operator. [*Note:* For an example of overloading the ++ and -- operators see
 16 §17.9.1. *end note*]

Operator notation	Functional notation
<i>op</i> x	operator <i>op</i> (x)
x <i>op</i>	operator <i>op</i> (x)
x <i>op</i> y	operator <i>op</i> (x, y)

18
 19 User-defined operator declarations always require at least one of the parameters to be of the class or struct
 20 type that contains the operator declaration. [*Note:* Thus, it is not possible for a user-defined operator to have
 21 the same signature as a predefined operator. *end note*]

22 User-defined operator declarations cannot modify the syntax, precedence, or associativity of an operator.
 23 [*Example:* For example, the / operator is always a binary operator, always has the precedence level
 24 specified in §14.2.1, and is always left-associative. *end example*]

25 [*Note:* While it is possible for a user-defined operator to perform any computation it pleases,
 26 implementations that produce results other than those that are intuitively expected are strongly discouraged.
 27 For example, an implementation of operator == should compare the two operands for equality and return
 28 an appropriate bool result. *end note*]

29 The descriptions of individual operators in §14.5 through §14.13 specify the predefined implementations of
 30 the operators and any additional rules that apply to each operator. The descriptions make use of the terms
 31 **unary operator overload resolution**, **binary operator overload resolution**, and **numeric promotion**,
 32 definitions of which are found in the following sections.

33 **14.2.3 Unary operator overload resolution**

34 An operation of the form *op* x or x *op*, where *op* is an overloadable unary operator, and x is an expression of
 35 type X, is processed as follows:

- 36 • The set of candidate user-defined operators provided by X for the operation operator *op*(x) is
 37 determined using the rules of §14.2.5.
- 38 • If the set of candidate user-defined operators is not empty, then this becomes the set of candidate
 39 operators for the operation. Otherwise, the predefined unary operator *op* implementations become the
 40 set of candidate operators for the operation. The predefined implementations of a given operator are
 41 specified in the description of the operator (§14.5 and §14.6).

- The overload resolution rules of §14.4.2 are applied to the set of candidate operators to select the best operator with respect to the argument list (x), and this operator becomes the result of the overload resolution process. If overload resolution fails to select a single best operator, a compile-time error occurs.

14.2.4 Binary operator overload resolution

An operation of the form $x \text{ op } y$, where op is an overloadable binary operator, x is an expression of type X , and y is an expression of type Y , is processed as follows:

- The set of candidate user-defined operators provided by X and Y for the operation `operator $op(x, y)$` is determined. The set consists of the union of the candidate operators provided by X and the candidate operators provided by Y , each determined using the rules of §14.2.5. If X and Y are the same type, or if X and Y are derived from a common base type, then shared candidate operators only occur in the combined set once.
- If the set of candidate user-defined operators is not empty, then this becomes the set of candidate operators for the operation. Otherwise, the predefined binary `operator op` implementations become the set of candidate operators for the operation. The predefined implementations of a given operator are specified in the description of the operator (§14.7 through §14.13).
- The overload resolution rules of §14.4.2 are applied to the set of candidate operators to select the best operator with respect to the argument list (x, y), and this operator becomes the result of the overload resolution process. If overload resolution fails to select a single best operator, a compile-time error occurs.

14.2.5 Candidate user-defined operators

Given a type T and an operation `operator $op(A)$` , where op is an overloadable operator and A is an argument list, the set of candidate user-defined operators provided by T for `operator $op(A)$` is determined as follows:

- For all `operator op` declarations in T , if at least one operator is applicable (§14.4.2.1) with respect to the argument list A , then the set of candidate operators consists of all applicable `operator op` declarations in T .
- Otherwise, if T is `object`, the set of candidate operators is empty.
- Otherwise, the set of candidate operators provided by T is the set of candidate operators provided by the direct base class of T .

14.2.6 Numeric promotions

This clause is informative.

Numeric promotion consists of automatically performing certain implicit conversions of the operands of the predefined unary and binary numeric operators. Numeric promotion is not a distinct mechanism, but rather an effect of applying overload resolution to the predefined operators. Numeric promotion specifically does not affect evaluation of user-defined operators, although user-defined operators can be implemented to exhibit similar effects.

As an example of numeric promotion, consider the predefined implementations of the binary `*` operator:

```
int operator *(int x, int y);
uint operator *(uint x, uint y);
long operator *(long x, long y);
ulong operator *(ulong x, ulong y);
float operator *(float x, float y);
double operator *(double x, double y);
decimal operator *(decimal x, decimal y);
```

When overload resolution rules (§14.4.2) are applied to this set of operators, the effect is to select the first of the operators for which implicit conversions exist from the operand types. [Example: For example, for the

1 operation `b * s`, where `b` is a `byte` and `s` is a `short`, overload resolution selects operator `*(int, int)`
2 as the best operator. Thus, the effect is that `b` and `s` are converted to `int`, and the type of the result is `int`.
3 Likewise, for the operation `i * d`, where `i` is an `int` and `d` is a `double`, overload resolution selects
4 operator `*(double, double)` as the best operator. *end example*

5 **End of informative text.**

6 14.2.6.1 Unary numeric promotions

7 **This clause is informative.**

8 Unary numeric promotion occurs for the operands of the predefined `+`, `-`, and `~` unary operators. Unary
9 numeric promotion simply consists of converting operands of type `sbyte`, `byte`, `short`, `ushort`, or `char`
10 to type `int`. Additionally, for the unary `-` operator, unary numeric promotion converts operands of type
11 `uint` to type `long`.

12 **End of informative text.**

13 14.2.6.2 Binary numeric promotions

14 **This clause is informative.**

15 Binary numeric promotion occurs for the operands of the predefined `+`, `-`, `*`, `/`, `%`, `&`, `|`, `^`, `==`, `!=`, `>`, `<`, `>=`,
16 and `<=` binary operators. Binary numeric promotion implicitly converts both operands to a common type
17 which, in case of the non-relational operators, also becomes the result type of the operation. Binary numeric
18 promotion consists of applying the following rules, in the order they appear here:

- 19 • If either operand is of type `decimal`, the other operand is converted to type `decimal`, or a compile-
20 time error occurs if the other operand is of type `float` or `double`.
- 21 • Otherwise, if either operand is of type `double`, the other operand is converted to type `double`.
- 22 • Otherwise, if either operand is of type `float`, the other operand is converted to type `float`.
- 23 • Otherwise, if either operand is of type `ulong`, the other operand is converted to type `ulong`, or a
24 compile-time error occurs if the other operand is of type `sbyte`, `short`, `int`, or `long`.
- 25 • Otherwise, if either operand is of type `long`, the other operand is converted to type `long`.
- 26 • Otherwise, if either operand is of type `uint` and the other operand is of type `sbyte`, `short`, or `int`,
27 both operands are converted to type `long`.
- 28 • Otherwise, if either operand is of type `uint`, the other operand is converted to type `uint`.
- 29 • Otherwise, both operands are converted to type `int`.

30 [*Note:* Note that the first rule disallows any operations that mix the `decimal` type with the `double` and
31 `float` types. The rule follows from the fact that there are no implicit conversions between the `decimal`
32 type and the `double` and `float` types. *end note*]

33 [*Note:* Also note that it is not possible for an operand to be of type `ulong` when the other operand is of a
34 signed integral type. The reason is that no integral type exists that can represent the full range of `ulong` as
35 well as the signed integral types. *end note*]

36 In both of the above cases, a cast expression can be used to explicitly convert one operand to a type that is
37 compatible with the other operand.

38 [*Example:* In the example

```
39     decimal AddPercent(decimal x, double percent) {  
40         return x * (1.0 + percent / 100.0);  
41     }
```

42 a compile-time error occurs because a `decimal` cannot be multiplied by a `double`. The error is resolved by
43 explicitly converting the second operand to `decimal`, as follows:

```

1     decimal AddPercent(decimal x, double percent) {
2         return x * (decimal)(1.0 + percent / 100.0);
3     }

```

4 *end example]*

5 **End of informative text.**

6 **14.3 Member lookup**

7 A member lookup is the process whereby the meaning of a name in the context of a type is determined. A
8 member lookup may occur as part of evaluating a *simple-name* (§14.5.2) or a *member-access* (§14.5.4) in an
9 expression.

10 A member lookup of a name *N* in a type *T* is processed as follows:

- 11 • First, the set of all accessible (§10.5) members named *N* declared in *T* and the base types (§14.3.1) of *T*
12 is constructed. Declarations that include an **override** modifier are excluded from the set. If no
13 members named *N* exist and are accessible, then the lookup produces no match, and the following steps
14 are not evaluated.
- 15 • Next, members that are hidden by other members are removed from the set. For every member *S.M* in
16 the set, where *S* is the type in which the member *M* is declared, the following rules are applied:
 - 17 ○ If *M* is a constant, field, property, event, type, or enumeration member, then all members declared in
18 a base type of *S* are removed from the set.
 - 19 ○ If *M* is a method, then all non-method members declared in a base type of *S* are removed from the
20 set, and all methods with the same signature as *M* declared in a base type of *S* are removed from the
21 set.
- 22 • Finally, having removed hidden members, the result of the lookup is determined:
 - 23 ○ If the set consists of a single non-method member, then this member is the result of the lookup.
 - 24 ○ Otherwise, if the set contains only methods, then this group of methods is the result of the lookup.
 - 25 ○ Otherwise, the lookup is ambiguous, and a compile-time error occurs (this situation can only occur
26 for a member lookup in an interface that has multiple direct base interfaces).

27 For member lookups in types other than interfaces, and member lookups in interfaces that are strictly single-
28 inheritance (each interface in the inheritance chain has exactly zero or one direct base interface), the effect of
29 the lookup rules is simply that derived members hide base members with the same name or signature. Such
30 single-inheritance lookups are never ambiguous. The ambiguities that can possibly arise from member
31 lookups in multiple-inheritance interfaces are described in §20.2.5.

32 **14.3.1 Base types**

33 For purposes of member lookup, a type *T* is considered to have the following base types:

- 34 • If *T* is **object**, then *T* has no base type.
- 35 • If *T* is a *value-type*, the base type of *T* is the class type **object**.
- 36 • If *T* is a *class-type*, the base types of *T* are the base classes of *T*, including the class type **object**.
- 37 • If *T* is an *interface-type*, the base types of *T* are the base interfaces of *T* and the class type **object**.
- 38 • If *T* is an *array-type*, the base types of *T* are the class types **System.Array** and **object**.
- 39 • If *T* is a *delegate-type*, the base types of *T* are the class types **System.Delegate** and **object**.

40 **14.4 Function members**

41 Function members are members that contain executable statements. Function members are always members
42 of types and cannot be members of namespaces. *C#* defines the following categories of function members:

- 1 • Methods
- 2 • Properties
- 3 • Events
- 4 • Indexers
- 5 • User-defined operators
- 6 • Instance constructors
- 7 • Static constructors
- 8 • Destructors

9 Except for static constructors and destructors (which cannot be invoked explicitly), the statements contained
 10 in function members are executed through function member invocations. The actual syntax for writing a
 11 function member invocation depends on the particular function member category.

12 The argument list (§14.4.1) of a function member invocation provides actual values or variable references
 13 for the parameters of the function member.

14 Invocations of methods, indexers, operators, and instance constructors employ overload resolution to
 15 determine which of a candidate set of function members to invoke. This process is described in §14.4.2.

16 Once a particular function member has been identified at compile-time, possibly through overload
 17 resolution, the actual run-time process of invoking the function member is described in §14.4.3.

18 [Note: The following table summarizes the processing that takes place in constructs involving the six
 19 categories of function members that can be explicitly invoked. In the table, *e*, *x*, *y*, and *value* indicate
 20 expressions classified as variables or values, *T* indicates an expression classified as a type, *F* is the simple
 21 name of a method, and *P* is the simple name of a property.

Construct	Example	Description
Method invocation	<i>F</i> (<i>x</i> , <i>y</i>)	Overload resolution is applied to select the best method <i>F</i> in the containing class or struct. The method is invoked with the argument list (<i>x</i> , <i>y</i>). If the method is not <code>static</code> , the instance expression is <code>this</code> .
	<i>T</i> . <i>F</i> (<i>x</i> , <i>y</i>)	Overload resolution is applied to select the best method <i>F</i> in the class or struct <i>T</i> . A compile-time error occurs if the method is not <code>static</code> . The method is invoked with the argument list (<i>x</i> , <i>y</i>).
	<i>e</i> . <i>F</i> (<i>x</i> , <i>y</i>)	Overload resolution is applied to select the best method <i>F</i> in the class, struct, or interface given by the type of <i>e</i> . A compile-time error occurs if the method is <code>static</code> . The method is invoked with the instance expression <i>e</i> and the argument list (<i>x</i> , <i>y</i>).
Property access	<i>P</i>	The <code>get</code> accessor of the property <i>P</i> in the containing class or struct is invoked. A compile-time error occurs if <i>P</i> is write-only. If <i>P</i> is not <code>static</code> , the instance expression is <code>this</code> .
	<i>P</i> = <i>value</i>	The <code>set</code> accessor of the property <i>P</i> in the containing class or struct is invoked with the argument list (<i>value</i>). A compile-time error occurs if <i>P</i> is read-only. If <i>P</i> is not <code>static</code> , the instance expression is <code>this</code> .
	<i>T</i> . <i>P</i>	The <code>get</code> accessor of the property <i>P</i> in the class or struct <i>T</i> is invoked. A compile-time error occurs if <i>P</i> is not <code>static</code> or if <i>P</i> is write-only.

Construct	Example	Description
	<code>T.P = value</code>	The <code>set</code> accessor of the property <code>P</code> in the class or struct <code>T</code> is invoked with the argument list (<code>value</code>). A compile-time error occurs if <code>P</code> is not <code>static</code> or if <code>P</code> is read-only.
	<code>e.P</code>	The <code>get</code> accessor of the property <code>P</code> in the class, struct, or interface given by the type of <code>e</code> is invoked with the instance expression <code>e</code> . A compile-time error occurs if <code>P</code> is <code>static</code> or if <code>P</code> is write-only.
	<code>e.P = value</code>	The <code>set</code> accessor of the property <code>P</code> in the class, struct, or interface given by the type of <code>e</code> is invoked with the instance expression <code>e</code> and the argument list (<code>value</code>). A compile-time error occurs if <code>P</code> is <code>static</code> or if <code>P</code> is read-only.
Event access	<code>E += value</code>	The <code>add</code> accessor of the event <code>E</code> in the containing class or struct is invoked. If <code>E</code> is not <code>static</code> , the instance expression is <code>this</code> .
	<code>E -= value</code>	The <code>remove</code> accessor of the event <code>E</code> in the containing class or struct is invoked. If <code>E</code> is not <code>static</code> , the instance expression is <code>this</code> .
	<code>T.E += value</code>	The <code>add</code> accessor of the event <code>E</code> in the class or struct <code>T</code> is invoked. A compile-time error occurs if <code>E</code> is not <code>static</code> .
	<code>T.E -= value</code>	The <code>remove</code> accessor of the event <code>E</code> in the class or struct <code>T</code> is invoked. A compile-time error occurs if <code>E</code> is not <code>static</code> .
	<code>e.E += value</code>	The <code>add</code> accessor of the event <code>E</code> in the class, struct, or interface given by the type of <code>e</code> is invoked with the instance expression <code>e</code> . A compile-time error occurs if <code>E</code> is <code>static</code> .
	<code>e.E -= value</code>	The <code>remove</code> accessor of the event <code>E</code> in the class, struct, or interface given by the type of <code>e</code> is invoked with the instance expression <code>e</code> . A compile-time error occurs if <code>E</code> is <code>static</code> .
Indexer access	<code>e[x, y]</code>	Overload resolution is applied to select the best indexer in the class, struct, or interface given by the type of <code>e</code> . The <code>get</code> accessor of the indexer is invoked with the instance expression <code>e</code> and the argument list (<code>x</code> , <code>y</code>). A compile-time error occurs if the indexer is write-only.
	<code>e[x, y] = value</code>	Overload resolution is applied to select the best indexer in the class, struct, or interface given by the type of <code>e</code> . The <code>set</code> accessor of the indexer is invoked with the instance expression <code>e</code> and the argument list (<code>x</code> , <code>y</code> , <code>value</code>). A compile-time error occurs if the indexer is read-only.
Operator invocation	<code>-x</code>	Overload resolution is applied to select the best unary operator in the class or struct given by the type of <code>x</code> . The selected operator is invoked with the argument list (<code>x</code>).
	<code>x + y</code>	Overload resolution is applied to select the best binary operator in the classes or structs given by the types of <code>x</code> and <code>y</code> . The selected operator is invoked with the argument list (<code>x</code> , <code>y</code>).
Instance constructor invocation	<code>new T(x, y)</code>	Overload resolution is applied to select the best instance constructor in the class or struct <code>T</code> . The instance constructor is invoked with the argument list (<code>x</code> , <code>y</code>).

1 *end note]*

1 14.4.1 Argument lists

2 Every function member invocation includes an argument list, which provides actual values or variable
3 references for the parameters of the function member. The syntax for specifying the argument list of a
4 function member invocation depends on the function member category:

- 5 • For instance constructors, methods, and delegates, the arguments are specified as an *argument-list*, as
6 described below.
- 7 • For properties, the argument list is empty when invoking the `get` accessor, and consists of the
8 expression specified as the right operand of the assignment operator when invoking the `set` accessor.
- 9 • For events, the argument list consists of the expression specified as the right operand of the `+=` or `-=`
10 operator.
- 11 • For indexers, the argument list consists of the expressions specified between the square brackets in the
12 indexer access. When invoking the `set` accessor, the argument list additionally includes the expression
13 specified as the right operand of the assignment operator.
- 14 • For user-defined operators, the argument list consists of the single operand of the unary operator or the
15 two operands of the binary operator.

16 The arguments of properties (§17.6), events (§17.7), indexers (§17.8), and user-defined operators (§17.9) are
17 always passed as value parameters (§17.5.1.1). Reference and output parameters are not supported for these
18 categories of function members.

19 The arguments of an instance constructor, method, or delegate invocation are specified as an *argument-list*:

```
20     argument-list:
21         argument
22         argument-list , argument
23
24     argument:
25         expression
26         ref variable-reference
27         out variable-reference
```

27 An *argument-list* consists of one or more *arguments*, separated by commas. Each argument can take one of
28 the following forms:

- 29 • An *expression*, indicating that the argument is passed as a value parameter (§17.5.1.1).
- 30 • The keyword `ref` followed by a *variable-reference* (§12.3.3), indicating that the argument is passed as a
31 reference parameter (§17.5.1.2). A variable must be definitely assigned (§12.3) before it can be passed
32 as a reference parameter. A volatile field (§17.4.3) cannot be passed as a reference parameter.
- 33 • The keyword `out` followed by a *variable-reference* (§12.3.3), indicating that the argument is passed as
34 an output parameter (§17.5.1.3). A variable is considered definitely assigned (§12.3) following a
35 function member invocation in which the variable is passed as an output parameter. A volatile field
36 (§17.4.3) cannot be passed as an output parameter.

37 During the run-time processing of a function member invocation (§14.4.3), the expressions or variable
38 references of an argument list are evaluated in order, from left to right, as follows:

- 39 • For a value parameter, the argument expression is evaluated and an implicit conversion (§13.1) to the
40 corresponding parameter type is performed. The resulting value becomes the initial value of the value
41 parameter in the function member invocation.
- 42 • For a reference or output parameter, the variable reference is evaluated and the resulting storage location
43 becomes the storage location represented by the parameter in the function member invocation. If the
44 variable reference given as a reference or output parameter is an array element of a *reference-type*, a
45 run-time check is performed to ensure that the element type of the array is identical to the type of the
46 parameter. If this check fails, a `System.ArrayTypeMismatchException` is thrown.

1 Methods, indexers, and instance constructors may declare their right-most parameter to be a parameter array
 2 (§17.5.1.4). Such function members are invoked either in their normal form or in their expanded form
 3 depending on which is applicable (§14.4.2.1):

- 4 • When a function member with a parameter array is invoked in its normal form, the argument given for
 5 the parameter array must be a single expression of a type that is implicitly convertible (§13.1) to the
 6 parameter array type. In this case, the parameter array acts precisely like a value parameter.
- 7 • When a function member with a parameter array is invoked in its expanded form, the invocation must
 8 specify zero or more arguments for the parameter array, where each argument is an expression of a type
 9 that is implicitly convertible (§13.1) to the element type of the parameter array. In this case, the
 10 invocation creates an instance of the parameter array type with a length corresponding to the number of
 11 arguments, initializes the elements of the array instance with the given argument values, and uses the
 12 newly created array instance as the actual argument.

13 The expressions of an argument list are always evaluated in the order they are written. [Example: Thus, the
 14 example

```

15     class Test
16     {
17         static void F(int x, int y, int z) {
18             System.Console.WriteLine("x = {0}, y = {1}, z = {2}", x, y, z);
19         }
20         static void Main() {
21             int i = 0;
22             F(i++, i++, i++);
23         }
24     }
  
```

25 produces the output

```

26     x = 0, y = 1, z = 2
  
```

27 *end example*]

28 The array covariance rules (§19.5) permit a value of an array type `A[]` to be a reference to an instance of an
 29 array type `B[]`, provided an implicit reference conversion exists from `B` to `A`. Because of these rules, when
 30 an array element of a *reference-type* is passed as a reference or output parameter, a run-time check is
 31 required to ensure that the actual element type of the array is *identical* to that of the parameter. [Example: In
 32 the example

```

33     class Test
34     {
35         static void F(ref object x) {...}
36         static void Main() {
37             object[] a = new object[10];
38             object[] b = new string[10];
39             F(ref a[0]);           // ok
40             F(ref b[1]);           // ArrayTypeMismatchException
41         }
42     }
  
```

43 the second invocation of `F` causes a `System.ArrayTypeMismatchException` to be thrown because the
 44 actual element type of `b` is `string` and not `object`. *end example*]

45 When a function member with a parameter array is invoked in its expanded form, the invocation is
 46 processed exactly as if an array creation expression with an array initializer (§14.5.10.2) was inserted around
 47 the expanded parameters. [Example: For example, given the declaration

```

48     void F(int x, int y, params object[] args);
  
```

49 the following invocations of the expanded form of the method

```

50     F(10, 20);
51     F(10, 20, 30, 40);
52     F(10, 20, 1, "hello", 3.0);
  
```

1 correspond exactly to

```
2     F(10, 20, new object[] {});  
3     F(10, 20, new object[] {30, 40});  
4     F(10, 20, new object[] {1, "hello", 3.0});
```

5 *end example*] In particular, note that an empty array is created when there are zero arguments given for the
6 parameter array.

7 **14.4.2 Overload resolution**

8 Overload resolution is a compile-time mechanism for selecting the best function member to invoke given an
9 argument list and a set of candidate function members. Overload resolution selects the function member to
10 invoke in the following distinct contexts within C#:

- 11 • Invocation of a method named in an *invocation-expression* (§14.5.5).
- 12 • Invocation of an instance constructor named in an *object-creation-expression* (§14.5.10.1).
- 13 • Invocation of an indexer accessor through an *element-access* (§14.5.6).
- 14 • Invocation of a predefined or user-defined operator referenced in an expression (§14.2.3 and §14.2.4).

15 Each of these contexts defines the set of candidate function members and the list of arguments in its own
16 unique way. However, once the candidate function members and the argument list have been identified, the
17 selection of the best function member is the same in all cases:

- 18 • First, the set of candidate function members is reduced to those function members that are applicable
19 with respect to the given argument list (§14.4.2.1). If this reduced set is empty, a compile-time error
20 occurs.
- 21 • Then, given the set of applicable candidate function members, the best function member in that set is
22 located. If the set contains only one function member, then that function member is the best function
23 member. Otherwise, the best function member is the one function member that is better than all other
24 function members with respect to the given argument list, provided that each function member is
25 compared to all other function members using the rules in §14.4.2.2. If there is not exactly one function
26 member that is better than all other function members, then the function member invocation is
27 ambiguous and a compile-time error occurs.

28 The following sections define the exact meanings of the terms *applicable function member* and *better*
29 *function member*.

30 **14.4.2.1 Applicable function member**

31 A function member is said to be an *applicable function member* with respect to an argument list A when all
32 of the following are true:

- 33 • The number of arguments in A is identical to the number of parameters in the function member
34 declaration.
- 35 • For each argument in A, the parameter passing mode of the argument (i.e., *value*, *ref*, or *out*) is
36 identical to the parameter passing mode of the corresponding parameter, and
 - 37 ○ for a *value* parameter or a parameter array, an implicit conversion (§13.1) exists from the type of the
38 argument to the type of the corresponding parameter, or
 - 39 ○ for a *ref* or *out* parameter, the type of the argument is identical to the type of the corresponding
40 parameter. [*Note*: After all, a *ref* or *out* parameter is an alias for the argument passed. *end note*]

41 For a function member that includes a parameter array, if the function member is applicable by the above
42 rules, it is said to be applicable in its *normal form*. If a function member that includes a parameter array is
43 not applicable in its normal form, the function member may instead be applicable in its *expanded form*:

- 1 • The expanded form is constructed by replacing the parameter array in the function member declaration
2 with zero or more value parameters of the element type of the parameter array such that the number of
3 arguments in the argument list *A* matches the total number of parameters. If *A* has fewer arguments than
4 the number of fixed parameters in the function member declaration, the expanded form of the function
5 member cannot be constructed and is thus not applicable.
- 6 • If the class, struct, or interface in which the function member is declared already contains another
7 applicable function member with the same signature as the expanded form, the expanded form is not
8 applicable.
- 9 • Otherwise, the expanded form is applicable if for each argument in *A* the parameter passing mode of the
10 argument is identical to the parameter passing mode of the corresponding parameter, and
 - 11 ○ for a fixed value parameter or a value parameter created by the expansion, an implicit conversion
12 (§13.1) exists from the type of the argument to the type of the corresponding parameter, or
 - 13 ○ for a `ref` or `out` parameter, the type of the argument is identical to the type of the corresponding
14 parameter.

15 14.4.2.2 Better function member

16 Given an argument list *A* with a set of argument types A_1, A_2, \dots, A_N and two applicable function members M_P
17 and M_Q with parameter types P_1, P_2, \dots, P_N and Q_1, Q_2, \dots, Q_N , M_P is defined to be a ***better function member***
18 than M_Q if

- 19 • for each argument, the implicit conversion from A_x to P_x is not worse than the implicit conversion from
20 A_x to Q_x , and
- 21 • for at least one argument, the conversion from A_x to P_x is better than the conversion from A_x to Q_x .

22 When performing this evaluation, if M_P or M_Q is applicable in its expanded form, then P_x or Q_x refers to a
23 parameter in the expanded form of the parameter list.

24 14.4.2.3 Better conversion

25 Given an implicit conversion C_1 that converts from a type S to a type T_1 , and an implicit conversion C_2 that
26 converts from a type S to a type T_2 , the ***better conversion*** of the two conversions is determined as follows:

- 27 • If T_1 and T_2 are the same type, neither conversion is better.
- 28 • If S is T_1 , C_1 is the better conversion.
- 29 • If S is T_2 , C_2 is the better conversion.
- 30 • If an implicit conversion from T_1 to T_2 exists, and no implicit conversion from T_2 to T_1 exists, C_1 is the
31 better conversion.
- 32 • If an implicit conversion from T_2 to T_1 exists, and no implicit conversion from T_1 to T_2 exists, C_2 is the
33 better conversion.
- 34 • If T_1 is `sbyte` and T_2 is `byte`, `ushort`, `uint`, or `ulong`, C_1 is the better conversion.
- 35 • If T_2 is `sbyte` and T_1 is `byte`, `ushort`, `uint`, or `ulong`, C_2 is the better conversion.
- 36 • If T_1 is `short` and T_2 is `ushort`, `uint`, or `ulong`, C_1 is the better conversion.
- 37 • If T_2 is `short` and T_1 is `ushort`, `uint`, or `ulong`, C_2 is the better conversion.
- 38 • If T_1 is `int` and T_2 is `uint`, or `ulong`, C_1 is the better conversion.
- 39 • If T_2 is `int` and T_1 is `uint`, or `ulong`, C_2 is the better conversion.
- 40 • If T_1 is `long` and T_2 is `ulong`, C_1 is the better conversion.
- 41 • If T_2 is `long` and T_1 is `ulong`, C_2 is the better conversion.

- 1 • Otherwise, neither conversion is better.

2 If an implicit conversion C_1 is defined by these rules to be a better conversion than an implicit conversion C_2 ,
3 then it is also the case that C_2 is a *worse conversion* than C_1 .

4 14.4.3 Function member invocation

5 This section describes the process that takes place at run-time to invoke a particular function member. It is
6 assumed that a compile-time process has already determined the particular member to invoke, possibly by
7 applying overload resolution to a set of candidate function members.

8 For purposes of describing the invocation process, function members are divided into two categories:

- 9 • Static function members. These are static methods, instance constructors, static property accessors, and
10 user-defined operators. Static function members are always non-virtual.
- 11 • Instance function members. These are instance methods, instance property accessors, and indexer
12 accessors. Instance function members are either non-virtual or virtual, and are always invoked on a
13 particular instance. The instance is computed by an instance expression, and it becomes accessible
14 within the function member as `this` (§14.5.7).

15 The run-time processing of a function member invocation consists of the following steps, where M is the
16 function member and, if M is an instance member, E is the instance expression:

- 17 • If M is a static function member:
- 18 ○ The argument list is evaluated as described in §14.4.1.
- 19 ○ M is invoked.
- 20 • If M is an instance function member declared in a *value-type*:
- 21 ○ E is evaluated. If this evaluation causes an exception, then no further steps are executed.
- 22 ○ If E is not classified as a variable, then a temporary local variable of E 's type is created and the value
23 of E is assigned to that variable. E is then reclassified as a reference to that temporary local variable.
24 The temporary variable is accessible as `this` within M , but not in any other way. Thus, only when E
25 is a true variable is it possible for the caller to observe the changes that M makes to `this`.
- 26 ○ The argument list is evaluated as described in §14.4.1.
- 27 ○ M is invoked. The variable referenced by E becomes the variable referenced by `this`.
- 28 • If M is an instance function member declared in a *reference-type*:
- 29 ○ E is evaluated. If this evaluation causes an exception, then no further steps are executed.
- 30 ○ The argument list is evaluated as described in §14.4.1.
- 31 ○ If the type of E is a *value-type*, a boxing conversion (§11.3.1) is performed to convert E to type
32 `object`, and E is considered to be of type `object` in the following steps. [*Note*: In this case, M
33 could only be a member of `System.Object`. *end note*]
- 34 ○ The value of E is checked to be valid. If the value of E is `null`, a
35 `System.NullReferenceException` is thrown and no further steps are executed.
- 36 ○ The function member implementation to invoke is determined:
- 37 • If the compile-time type of E is an interface, the function member to invoke is the
38 implementation of M provided by the run-time type of the instance referenced by E . This
39 function member is determined by applying the interface mapping rules (§20.4.2) to determine
40 the implementation of M provided by the run-time type of the instance referenced by E .
- 41 • Otherwise, if M is a virtual function member, the function member to invoke is the
42 implementation of M provided by the run-time type of the instance referenced by E . This

1 function member is determined by applying the rules for determining the most derived
 2 implementation (§17.5.3) of *M* with respect to the run-time type of the instance referenced by *E*.

- 3 • Otherwise, *M* is a non-virtual function member, and the function member to invoke is *M* itself.
- 4 ○ The function member implementation determined in the step above is invoked. The object
 5 referenced by *E* becomes the object referenced by `this`.

6 14.4.3.1 Invocations on boxed instances

7 A function member implemented in a *value-type* can be invoked through a boxed instance of that *value-type*
 8 in the following situations:

- 9 • When the function member is an `override` of a method inherited from type `object` and is invoked
 10 through an instance expression of type `object`.
- 11 • When the function member is an implementation of an interface function member and is invoked
 12 through an instance expression of an *interface-type*.
- 13 • When the function member is invoked through a delegate.

14 In these situations, the boxed instance is considered to contain a variable of the *value-type*, and this variable
 15 becomes the variable referenced by `this` within the function member invocation. [*Note:* In particular, this
 16 means that when a function member is invoked on a boxed instance, it is possible for the function member to
 17 modify the value contained in the boxed instance. *end note*]

18 14.5 Primary expressions

19 Primary expressions include the simplest forms of expressions.

20 *primary-expression:*

21 *array-creation-expression*

22 *primary-no-array-creation-expression*

23 *primary-no-array-creation-expression:*

24 *literal*

25 *simple-name*

26 *parenthesized-expression*

27 *member-access*

28 *invocation-expression*

29 *element-access*

30 *this-access*

31 *base-access*

32 *post-increment-expression*

33 *post-decrement-expression*

34 *object-creation-expression*

35 *delegate-creation-expression*

36 *typeof-expression*

37 *sizeof-expression*

38 *checked-expression*

39 *unchecked-expression*

40 Primary expressions are divided between *array-creation-expressions* and *primary-no-array-creation-*
 41 *expressions*. Treating *array-creation-expression* in this way, rather than listing it along with the other simple
 42 expression forms, enables the grammar to disallow potentially confusing code such as

43 `object o = new int[3][1];`

44 which would otherwise be interpreted as

45 `object o = (new int[3])[1];`

1 **14.5.1 Literals**

2 A *primary-expression* that consists of a *literal* (§9.4.4) is classified as a value.

3 **14.5.2 Simple names**

4 A *simple-name* consists of a single identifier.

5 *simple-name*:
6 *identifier*

7 A *simple-name* is evaluated and classified as follows:

- 8 • If the *simple-name* appears within a *block* and if the *block*'s (or an enclosing *block*'s) local variable
9 declaration space (§10.3) contains a local variable or parameter with the given name, then the *simple-*
10 *name* refers to that local variable or parameter and is classified as a variable.
- 11 • Otherwise, for each type *T*, starting with the immediately enclosing class, struct, or enumeration
12 declaration and continuing with each enclosing outer class or struct declaration (if any), if a member
13 lookup of the *simple-name* in *T* produces a match:
 - 14 ○ If *T* is the immediately enclosing class or struct type and the lookup identifies one or more methods,
15 the result is a method group with an associated instance expression of `this`.
 - 16 ○ If *T* is the immediately enclosing class or struct type, if the lookup identifies an instance member,
17 and if the reference occurs within the *block* of an instance constructor, an instance method, or an
18 instance accessor, the result is the same as a member access (§14.5.4) of the form `this.E`, where *E*
19 is the *simple-name*.
 - 20 ○ Otherwise, the result is the same as a member access (§14.5.4) of the form `T.E`, where *E* is the
21 *simple-name*. In this case, it is a compile-time error for the *simple-name* to refer to an instance
22 member.
- 23 • Otherwise, starting with the namespace in which the *simple-name* occurs, continuing with each
24 enclosing namespace (if any), and ending with the global namespace, the following steps are evaluated
25 until an entity is located:
 - 26 ○ If the namespace contains a namespace member with the given name, then the *simple-name* refers to
27 that member and, depending on the member, is classified as a namespace or a type.
 - 28 ○ Otherwise, if the namespace has a corresponding namespace declaration enclosing the location
29 where the *simple-name* occurs, then:
 - 30 • If the namespace declaration contains a *using-alias-directive* that associates the given name with
31 an imported namespace or type, then the *simple-name* refers to that namespace or type.
 - 32 • Otherwise, if the namespaces imported by the *using-namespace-directives* of the namespace
33 declaration contain exactly one type with the given name, then the *simple-name* refers to that
34 type.
 - 35 • Otherwise, if the namespaces imported by the *using-namespace-directives* of the namespace
36 declaration contain more than one type with the given name, then the *simple-name* is ambiguous
37 and a compile-time error occurs.
 - 38 • Otherwise, the name given by the *simple-name* is undefined and a compile-time error occurs.

39 **14.5.2.1 Invariant meaning in blocks**

40 For each occurrence of a given identifier as a *simple-name* in an expression, every other occurrence of the
41 same identifier as a *simple-name* in an expression within the immediately enclosing *block* (§15.2) or *switch-*
42 *block* (§15.7.2) must refer to the same entity. This rule ensures that the meaning of a name in the context of
43 an expression is always the same within a block.

44 The example


```

1      class Test
2      {
3          double x;
4          void F(bool b) {
5              x = 1.0;
6              if (b) {
7                  int x = 1;
8              }
9          }
10     }

```

11 results in a compile-time error because `x` refers to different entities within the outer block (the extent of
12 which includes the nested block in the `if` statement). In contrast, the example

```

13     class Test
14     {
15         double x;
16         void F(bool b) {
17             if (b) {
18                 x = 1.0;
19             }
20             else {
21                 int x = 1;
22             }
23         }
24     }

```

25 is permitted because the name `x` is never used in the outer block.

26 Note that the rule of invariant meaning applies only to simple names. It is perfectly valid for the same
27 identifier to have one meaning as a simple name and another meaning as right operand of a member access
28 (§14.5.4). [*Example:* For example:

```

29     struct Point
30     {
31         int x, y;
32         public Point(int x, int y) {
33             this.x = x;
34             this.y = y;
35         }
36     }

```

37 The example above illustrates a common pattern of using the names of fields as parameter names in an
38 instance constructor. In the example, the simple names `x` and `y` refer to the parameters, but that does not
39 prevent the member access expressions `this.x` and `this.y` from accessing the fields. *end example*]

40 14.5.3 Parenthesized expressions

41 A *parenthesized-expression* consists of an *expression* enclosed in parentheses.

```

42     parenthesized-expression:
43     ( expression )

```

44 A *parenthesized-expression* is evaluated by evaluating the *expression* within the parentheses. If the
45 *expression* within the parentheses denotes a namespace, type, or method group, a compile-time error occurs.
46 Otherwise, the result of the *parenthesized-expression* is the result of the evaluation of the contained
47 *expression*.

48 14.5.4 Member access

49 A *member-access* consists of a *primary-expression* or a *predefined-type*, followed by a “.” token, followed
50 by an *identifier*.

1 *member-access*:

2 *primary-expression* . *identifier*

3 *predefined-type* . *identifier*

4 *predefined-type*: one of

5 bool byte char decimal double float int long

6 object sbyte short string uint ulong ushort

7 A *member-access* of the form *E* . *I*, where *E* is a *primary-expression* or a *predefined-type* and *I* is an
8 *identifier*, is evaluated and classified as follows:

- 9 • If *E* is a namespace and *I* is the name of an accessible member of that namespace, then the result is that
10 member and, depending on the member, is classified as a namespace or a type.
- 11 • If *E* is a *predefined-type* or a *primary-expression* classified as a type, and a member lookup (§14.3) of *I*
12 in *E* produces a match, then *E* . *I* is evaluated and classified as follows:
 - 13 ○ If *I* identifies a type, then the result is that type.
 - 14 ○ If *I* identifies one or more methods, then the result is a method group with no associated instance
15 expression.
 - 16 ○ If *I* identifies a `static` property, then the result is a property access with no associated instance
17 expression.
 - 18 ○ If *I* identifies a `static` field:
 - 19 • If the field is `readonly` and the reference occurs outside the static constructor of the class or
20 struct in which the field is declared, then the result is a value, namely the value of the static field
21 *I* in *E*.
 - 22 • Otherwise, the result is a variable, namely the static field *I* in *E*.
 - 23 ○ If *I* identifies a `static` event:
 - 24 • If the reference occurs within the class or struct in which the event is declared, and the event
25 was declared without *event-accessor-declarations* (§17.7), then *E* . *I* is processed exactly as if *I*
26 was a static field.
 - 27 • Otherwise, the result is an event access with no associated instance expression.
 - 28 ○ If *I* identifies a constant, then the result is a value, namely the value of that constant.
 - 29 ○ If *I* identifies an enumeration member, then the result is a value, namely the value of that
30 enumeration member.
 - 31 ○ Otherwise, *E* . *I* is an invalid member reference, and a compile-time error occurs.
- 32 • If *E* is a property access, indexer access, variable, or value, the type of which is *T*, and a member lookup
33 (§14.3) of *I* in *T* produces a match, then *E* . *I* is evaluated and classified as follows:
 - 34 ○ First, if *E* is a property or indexer access, then the value of the property or indexer access is obtained
35 (§14.1.1) and *E* is reclassified as a value.
 - 36 ○ If *I* identifies one or more methods, then the result is a method group with an associated instance
37 expression of *E*.
 - 38 ○ If *I* identifies an instance property, then the result is a property access with an associated instance
39 expression of *E*.
 - 40 ○ If *T* is a *class-type* and *I* identifies an instance field of that *class-type*:
 - 41 • If the value of *E* is `null`, then a `System.NullReferenceException` is thrown.

- 1 • Otherwise, if the field is `readonly` and the reference occurs outside an instance constructor of
2 the class in which the field is declared, then the result is a value, namely the value of the field `I`
3 in the object referenced by `E`.
- 4 • Otherwise, the result is a variable, namely the field `I` in the object referenced by `E`.
- 5 ○ If `T` is a *struct-type* and `I` identifies an instance field of that *struct-type*:
 - 6 • If `E` is a value, or if the field is `readonly` and the reference occurs outside an instance
7 constructor of the struct in which the field is declared, then the result is a value, namely the
8 value of the field `I` in the struct instance given by `E`.
 - 9 • Otherwise, the result is a variable, namely the field `I` in the struct instance given by `E`.
- 10 ○ If `I` identifies an instance event:
 - 11 • If the reference occurs within the class or struct in which the event is declared, and the event
12 was declared without *event-accessor-declarations* (§17.7), then `E.I` is processed exactly as if `I`
13 was an instance field.
 - 14 • Otherwise, the result is an event access with an associated instance expression of `E`.
- 15 • Otherwise, `E.I` is an invalid member reference, and a compile-time error occurs.

16 14.5.4.1 Identical simple names and type names

17 In a member access of the form `E.I`, if `E` is a single identifier, and if the meaning of `E` as a *simple-name*
18 (§14.5.2) is a constant, field, property, local variable, or parameter with the same type as the meaning of `E` as
19 a *type-name* (§10.8), then both possible meanings of `E` are permitted. The two possible meanings of `E.I` are
20 never ambiguous, since `I` must necessarily be a member of the type `E` in both cases. In other words, the rule
21 simply permits access to the static members of `E` where a compile-time error would otherwise have occurred.
22 [*Example:* For example:

```

23     struct Color
24     {
25         public static readonly Color white = new Color(...);
26         public static readonly Color Black = new Color(...);
27         public Color Complement() {...}
28     }
29     class A
30     {
31         public color color;           // Field color of type Color
32         void F() {
33             Color = color.Black;     // References Color.Black static
34             member
35             Color = Color.Complement(); // Invokes Complement() on Color
36             field
37         }
38         static void G() {
39             Color c = Color.white;   // References Color.white static
40             member
41         }
42     }

```

43 Within the `A` class, those occurrences of the `Color` identifier that reference the `Color` type are underlined,
44 and those that reference the `Color` field are not underlined. *end example*]

45 14.5.5 Invocation expressions

46 An *invocation-expression* is used to invoke a method.

47 *invocation-expression:*
48 *primary-expression* (*argument-list_{opt}*)

1 The *primary-expression* of an *invocation-expression* must be a method group or a value of a *delegate-type*.
2 If the *primary-expression* is a method group, the *invocation-expression* is a method invocation (§14.5.5.1). If
3 the *primary-expression* is a value of a *delegate-type*, the *invocation-expression* is a delegate invocation
4 (§14.5.5.2). If the *primary-expression* is neither a method group nor a value of a *delegate-type*, a compile-
5 time error occurs.

6 The optional *argument-list* (§14.4.1) provides values or variable references for the parameters of the method.

7 The result of evaluating an *invocation-expression* is classified as follows:

- 8 • If the *invocation-expression* invokes a method or delegate that returns `void`, the result is nothing. An
9 expression that is classified as nothing cannot be an operand of any operator, and is permitted only in the
10 context of a *statement-expression* (§15.6).
- 11 • Otherwise, the result is a value of the type returned by the method or delegate.

12 14.5.5.1 Method invocations

13 For a method invocation, the *primary-expression* of the *invocation-expression* must be a method group. The
14 method group identifies the one method to invoke or the set of overloaded methods from which to choose a
15 specific method to invoke. In the latter case, determination of the specific method to invoke is based on the
16 context provided by the types of the arguments in the *argument-list*.

17 The compile-time processing of a method invocation of the form $M(A)$, where M is a method group and A is
18 an optional *argument-list*, consists of the following steps:

- 19 • The set of candidate methods for the method invocation is constructed. Starting with the set of methods
20 associated with M , which were found by a previous member lookup (§14.3), the set is reduced to those
21 methods that are applicable with respect to the argument list A . The set reduction consists of applying
22 the following rules to each method $T.N$ in the set, where T is the type in which the method N is declared:
 - 23 ○ If N is not applicable with respect to A (§14.4.2.1), then N is removed from the set.
 - 24 ○ If N is applicable with respect to A (§14.4.2.1), then all methods declared in a base type of T are
25 removed from the set.
- 26 • If the resulting set of candidate methods is empty, then no applicable methods exist, and a compile-time
27 error occurs. If the candidate methods are not all declared in the same type, the method invocation is
28 ambiguous, and a compile-time error occurs (this latter situation can only occur for an invocation of a
29 method in an interface that has multiple direct base interfaces, as described in §20.2.5).
- 30 • The best method of the set of candidate methods is identified using the overload resolution rules of
31 §14.4.2. If a single best method cannot be identified, the method invocation is ambiguous, and a
32 compile-time error occurs.
- 33 • Given a best method, the invocation of the method is validated in the context of the method group: If the
34 best method is a static method, the method group must have resulted from a *simple-name* or a *member-*
35 *access* through a type. If the best method is an instance method, the method group must have resulted
36 from a *simple-name*, a *member-access* through a variable or value, or a *base-access*. If neither of these
37 requirements are true, a compile-time error occurs.

38 Once a method has been selected and validated at compile-time by the above steps, the actual run-time
39 invocation is processed according to the rules of function member invocation described in §14.4.3.

40 [Note: The intuitive effect of the resolution rules described above is as follows: To locate the particular
41 method invoked by a method invocation, start with the type indicated by the method invocation and proceed
42 up the inheritance chain until at least one applicable, accessible, non-override method declaration is found.
43 Then perform overload resolution on the set of applicable, accessible, non-override methods declared in that
44 type and invoke the method thus selected. *end note*]

1 14.5.5.2 Delegate invocations

2 For a delegate invocation, the *primary-expression* of the *invocation-expression* must be a value of a
3 *delegate-type*. Furthermore, considering the *delegate-type* to be a function member with the same parameter
4 list as the *delegate-type*, the *delegate-type* must be applicable (§14.4.2.1) with respect to the *argument-list* of
5 the *invocation-expression*.

6 The run-time processing of a delegate invocation of the form $D(A)$, where D is a *primary-expression* of a
7 *delegate-type* and A is an optional *argument-list*, consists of the following steps:

- 8 • D is evaluated. If this evaluation causes an exception, no further steps are executed.
- 9 • The value of D is checked to be valid. If the value of D is `null`, a
10 `System.NullReferenceException` is thrown and no further steps are executed.
- 11 • Otherwise, D is a reference to a delegate instance. A function member invocation (§14.4.3) is performed
12 on the method referenced by the delegate. If the method is an instance method, the instance of the
13 invocation becomes the instance referenced by the delegate.

14 14.5.6 Element access

15 An *element-access* consists of a *primary-no-array-creation-expression*, followed by a “[“ token, followed
16 by an *expression-list*, followed by a “]” token. The *expression-list* consists of one or more *expressions*,
17 separated by commas.

18 *element-access*:

19 *primary-no-array-creation-expression* [*expression-list*]

20 *expression-list*:

21 *expression*

22 *expression-list* , *expression*

23 If the *primary-no-array-creation-expression* of an *element-access* is a value of an *array-type*, the *element-*
24 *access* is an array access (§14.5.6.1). Otherwise, the *primary-no-array-creation-expression* must be a
25 variable or value of a class, struct, or interface type that has one or more indexer members, in which case the
26 *element-access* is an indexer access (§14.5.6.2).

27 14.5.6.1 Array access

28 For an array access, the *primary-no-array-creation-expression* of the *element-access* must be a value of an
29 *array-type*. The number of expressions in the *expression-list* must be the same as the rank of the *array-type*,
30 and each expression must be of type `int`, `uint`, `long`, `ulong`, or of a type that can be implicitly converted
31 to one or more of these types.

32 The result of evaluating an array access is a variable of the element type of the array, namely the array
33 element selected by the value(s) of the expression(s) in the *expression-list*.

34 The run-time processing of an array access of the form $P[A]$, where P is a *primary-no-array-creation-*
35 *expression* of an *array-type* and A is an *expression-list*, consists of the following steps:

- 36 • P is evaluated. If this evaluation causes an exception, no further steps are executed.
- 37 • The index expressions of the *expression-list* are evaluated in order, from left to right. Following
38 evaluation of each index expression, an implicit conversion (§13.1) to one of the following types is
39 performed: `int`, `uint`, `long`, `ulong`. The first type in this list for which an implicit conversion exists is
40 chosen. For instance, if the index expression is of type `short` then an implicit conversion to `int` is
41 performed, since implicit conversions from `short` to `int` and from `short` to `long` are possible. If
42 evaluation of an index expression or the subsequent implicit conversion causes an exception, then no
43 further index expressions are evaluated and no further steps are executed.
- 44 • The value of P is checked to be valid. If the value of P is `null`, a
45 `System.NullReferenceException` is thrown and no further steps are executed.

- 1 • The value of each expression in the *expression-list* is checked against the actual bounds of each
2 dimension of the array instance referenced by P. If one or more values are out of range, a
3 `System.IndexOutOfRangeException` is thrown and no further steps are executed.
- 4 • The location of the array element given by the index expression(s) is computed, and this location
5 becomes the result of the array access.

6 14.5.6.2 Indexer access

7 For an indexer access, the *primary-no-array-creation-expression* of the *element-access* must be a variable
8 or value of a class, struct, or interface type, and this type must implement one or more indexers that are
9 applicable with respect to the *expression-list* of the *element-access*.

10 The compile-time processing of an indexer access of the form `P[A]`, where P is a *primary-no-array-*
11 *creation-expression* of a class, struct, or interface type T, and A is an *expression-list*, consists of the
12 following steps:

- 13 • The set of indexers provided by T is constructed. The set consists of all indexers declared in T or a base
14 type of T that are not `override` declarations and are accessible in the current context (§10.5).
- 15 • The set is reduced to those indexers that are applicable and not hidden by other indexers. The following
16 rules are applied to each indexer S.I in the set, where S is the type in which the indexer I is declared:
 - 17 ○ If I is not applicable with respect to A (§14.4.2.1), then I is removed from the set.
 - 18 ○ If I is applicable with respect to A (§14.4.2.1), then all indexers declared in a base type of S are
19 removed from the set.
- 20 • If the resulting set of candidate indexers is empty, then no applicable indexers exist, and a compile-time
21 error occurs. If the candidate indexers are not all declared in the same type, the indexer access is
22 ambiguous, and a compile-time error occurs (this latter situation can only occur for an indexer access on
23 an instance of an interface that has multiple direct base interfaces).
- 24 • The best indexer of the set of candidate indexers is identified using the overload resolution rules of
25 §14.4.2. If a single best indexer cannot be identified, the indexer access is ambiguous, and a compile-
26 time error occurs.
- 27 • The index expressions of the *expression-list* are evaluated in order, from left to right. The result of
28 processing the indexer access is an expression classified as an indexer access. The indexer access
29 expression references the indexer determined in the step above, and has an associated instance
30 expression of P and an associated argument list of A.

31 Depending on the context in which it is used, an indexer access causes invocation of either the *get-accessor*
32 or the *set-accessor* of the indexer. If the indexer access is the target of an assignment, the *set-accessor* is
33 invoked to assign a new value (§14.13.1). In all other cases, the *get-accessor* is invoked to obtain the current
34 value (§14.1.1).

35 14.5.7 This access

36 A *this-access* consists of the reserved word `this`.

37 *this-access*:
38 `this`

39 A *this-access* is permitted only in the *block* of an instance constructor, an instance method, or an instance
40 accessor. It has one of the following meanings:

- 41 • When `this` is used in a *primary-expression* within an instance constructor of a class, it is classified as a
42 value. The type of the value is the class within which the usage occurs, and the value is a reference to the
43 object being constructed.

- When `this` is used in a *primary-expression* within an instance method or instance accessor of a class, it is classified as a value. The type of the value is the class within which the usage occurs, and the value is a reference to the object for which the method or accessor was invoked.
- When `this` is used in a *primary-expression* within an instance constructor of a struct, it is classified as a variable. The type of the variable is the struct within which the usage occurs, and the variable represents the struct being constructed. The `this` variable of an instance constructor of a struct behaves exactly the same as an `out` parameter of the struct type—in particular, this means that the variable must be definitely assigned in every execution path of the instance constructor.
- When `this` is used in a *primary-expression* within an instance method or instance accessor of a struct, it is classified as a variable. The type of the variable is the struct within which the usage occurs, and the variable represents the struct for which the method or accessor was invoked. The `this` variable of an instance method of a struct behaves exactly the same as a `ref` parameter of the struct type.

Use of `this` in a *primary-expression* in a context other than the ones listed above is a compile-time error. In particular, it is not possible to refer to `this` in a static method, a static property accessor, or in a *variable-initializer* of a field declaration.

14.5.8 Base access

A *base-access* consists of the reserved word `base` followed by either a “.” token and an identifier or an *expression-list* enclosed in square brackets:

```
base-access:
    base . identifier
    base [ expression-list ]
```

A *base-access* is used to access base class members that are hidden by similarly named members in the current class or struct. A *base-access* is permitted only in the *block* of an instance constructor, an instance method, or an instance accessor. When `base.I` occurs in a class or struct, `I` must denote a member of the base class of that class or struct. Likewise, when `base[E]` occurs in a class, an applicable indexer must exist in the base class.

At compile-time, *base-access* expressions of the form `base.I` and `base[E]` are evaluated exactly as if they were written `((B)this).I` and `((B)this)[E]`, where `B` is the base class of the class or struct in which the construct occurs. Thus, `base.I` and `base[E]` correspond to `this.I` and `this[E]`, except `this` is viewed as an instance of the base class.

When a *base-access* references a virtual function member (a method, property, or indexer), the determination of which function member to invoke at run-time (§14.4.3) is changed. The function member that is invoked is determined by finding the most derived implementation (§17.5.3) of the function member with respect to `B` (instead of with respect to the run-time type of `this`, as would be usual in a non-base access). Thus, within an `override` of a `virtual` function member, a *base-access* can be used to invoke the inherited implementation of the function member. If the function member referenced by a *base-access* is abstract, a compile-time error occurs.

14.5.9 Postfix increment and decrement operators

```
post-increment-expression:
    primary-expression ++

post-decrement-expression:
    primary-expression --
```

The operand of a postfix increment or decrement operation must be an expression classified as a variable, a property access, or an indexer access. The result of the operation is a value of the same type as the operand.

If the operand of a postfix increment or decrement operation is a property or indexer access, the property or indexer must have both a `get` and a `set` accessor. If this is not the case, a compile-time error occurs.

1 Unary operator overload resolution (§14.2.3) is applied to select a specific operator implementation.
2 Predefined ++ and -- operators exist for the following types: `sbyte`, `byte`, `short`, `ushort`, `int`, `uint`,
3 `long`, `ulong`, `char`, `float`, `double`, `decimal`, and any enum type. The predefined ++ operators return the
4 value produced by adding 1 to the operand, and the predefined -- operators return the value produced by
5 subtracting 1 from the operand.

6 The run-time processing of a postfix increment or decrement operation of the form `x++` or `x--` consists of
7 the following steps:

- 8 • If `x` is classified as a variable:
 - 9 ○ `x` is evaluated to produce the variable.
 - 10 ○ The value of `x` is saved.
 - 11 ○ The selected operator is invoked with the saved value of `x` as its argument.
 - 12 ○ The value returned by the operator is stored in the location given by the evaluation of `x`.
 - 13 ○ The saved value of `x` becomes the result of the operation.
- 14 • If `x` is classified as a property or indexer access:
 - 15 ○ The instance expression (if `x` is not `static`) and the argument list (if `x` is an indexer access)
16 associated with `x` are evaluated, and the results are used in the subsequent `get` and `set` accessor
17 invocations.
 - 18 ○ The `get` accessor of `x` is invoked and the returned value is saved.
 - 19 ○ The selected operator is invoked with the saved value of `x` as its argument.
 - 20 ○ The `set` accessor of `x` is invoked with the value returned by the operator as its `value` argument.
 - 21 ○ The saved value of `x` becomes the result of the operation.

22 The ++ and -- operators also support prefix notation (§14.6.5). The result of `x++` or `x--` is the value of `x`
23 *before* the operation, whereas the result of `++x` or `--x` is the value of `x` *after* the operation. In either case, `x`
24 itself has the same value after the operation.

25 An operator ++ or operator -- implementation can be invoked using either postfix or prefix notation.
26 It is not possible to have separate operator implementations for the two notations.

27 14.5.10 The new operator

28 The `new` operator is used to create new instances of types.

29 There are three forms of `new` expressions:

- 30 • Object creation expressions are used to create new instances of class types and value types.
- 31 • Array creation expressions are used to create new instances of array types.
- 32 • Delegate creation expressions are used to create new instances of delegate types.

33 The `new` operator implies creation of an instance of a type, but does not necessarily imply dynamic
34 allocation of memory. In particular, instances of value types require no additional memory beyond the
35 variables in which they reside, and no dynamic allocations occur when `new` is used to create instances of
36 value types.

37 14.5.10.1 Object creation expressions

38 An *object-creation-expression* is used to create a new instance of a *class-type* or a *value-type*.

39 *object-creation-expression*:
40 `new type (argument-listopt)`

1 The *type* of an *object-creation-expression* must be a *class-type* or a *value-type*. The *type* cannot be an
 2 **abstract class-type**.

3 The optional *argument-list* (§14.4.1) is permitted only if the *type* is a *class-type* or a *struct-type*.

4 The compile-time processing of an *object-creation-expression* of the form `new T(A)`, where `T` is a *class-type*
 5 or a *value-type* and `A` is an optional *argument-list*, consists of the following steps:

- 6 • If `T` is a *value-type* and `A` is not present:
 - 7 ○ The *object-creation-expression* is a default constructor invocation. The result of the *object-creation-*
 8 *expression* is a value of type `T`, namely the default value for `T` as defined in §11.1.1.
- 9 • Otherwise, if `T` is a *class-type* or a *struct-type*:
 - 10 ○ If `T` is an **abstract class-type**, a compile-time error occurs.
 - 11 ○ The instance constructor to invoke is determined using the overload resolution rules of §14.4.2. The
 12 set of candidate instance constructors consists of all accessible instance constructors declared in `T`. If
 13 the set of candidate instance constructors is empty, or if a single best instance constructor cannot be
 14 identified, a compile-time error occurs.
 - 15 ○ The result of the *object-creation-expression* is a value of type `T`, namely the value produced by
 16 invoking the instance constructor determined in the step above.
- 17 • Otherwise, the *object-creation-expression* is invalid, and a compile-time error occurs.

18 The run-time processing of an *object-creation-expression* of the form `new T(A)`, where `T` is *class-type* or a
 19 *struct-type* and `A` is an optional *argument-list*, consists of the following steps:

- 20 • If `T` is a *class-type*:
 - 21 ○ A new instance of class `T` is allocated. If there is not enough memory available to allocate the new
 22 instance, a `System.OutOfMemoryException` is thrown and no further steps are executed.
 - 23 ○ All fields of the new instance are initialized to their default values (§12.2).
 - 24 ○ The instance constructor is invoked according to the rules of function member invocation (§14.4.3).
 25 A reference to the newly allocated instance is automatically passed to the instance constructor and
 26 the instance can be accessed from within that constructor as `this`.
- 27 • If `T` is a *struct-type*:
 - 28 ○ An instance of type `T` is created by allocating a temporary local variable. Since an instance
 29 constructor of a *struct-type* is required to definitely assign a value to each field of the instance being
 30 created, no initialization of the temporary variable is necessary.
 - 31 ○ The instance constructor is invoked according to the rules of function member invocation (§14.4.3).
 32 A reference to the newly allocated instance is automatically passed to the instance constructor and
 33 the instance can be accessed from within that constructor as `this`.

34 14.5.10.2 Array creation expressions

35 An *array-creation-expression* is used to create a new instance of an *array-type*.

36 *array-creation-expression*:

37 `new non-array-type [expression-list] rank-specifiersopt array-initializeropt`
 38 `new array-type array-initializer`

39 An array creation expression of the first form allocates an array instance of the type that results from
 40 deleting each of the individual expressions from the expression list. For example, the array creation
 41 expression `new int[10,20]` produces an array instance of type `int[,]`, and the array creation expression
 42 `new int[10][,]` produces an array of type `int[][,]`. Each expression in the expression list must be of
 43 type `int`, `uint`, `long`, or `ulong`, or of a type that can be implicitly converted to one or more of these types.
 44 The value of each expression determines the length of the corresponding dimension in the newly allocated

1 array instance. Since the length of an array dimension must be nonnegative, it is a compile-time error to
2 have a constant expression with a negative value, in the expression list.

3 Except in an unsafe context (§25.1), the layout of arrays is unspecified.

4 If an array creation expression of the first form includes an array initializer, each expression in the
5 expression list must be a constant and the rank and dimension lengths specified by the expression list must
6 match those of the array initializer.

7 In an array creation expression of the second form, the rank of the specified array type must match that of
8 the array initializer. The individual dimension lengths are inferred from the number of elements in each of
9 the corresponding nesting levels of the array initializer. Thus, the expression

```
10     new int[,] {{0, 1}, {2, 3}, {4, 5}}
```

11 exactly corresponds to

```
12     new int[3, 2] {{0, 1}, {2, 3}, {4, 5}}
```

13 Array initializers are described further in §19.6.

14 The result of evaluating an array creation expression is classified as a value, namely a reference to the newly
15 allocated array instance. The run-time processing of an array creation expression consists of the following
16 steps:

- 17 • The dimension length expressions of the *expression-list* are evaluated in order, from left to right.
18 Following evaluation of each expression, an implicit conversion (§13.1) to one of the following types is
19 performed: `int`, `uint`, `long`, `ulong`. The first type in this list for which an implicit conversion exists is
20 chosen. If evaluation of an expression or the subsequent implicit conversion causes an exception, then
21 no further expressions are evaluated and no further steps are executed.
- 22 • The computed values for the dimension lengths are validated, as follows: If one or more of the values
23 are less than zero, a `System.OverflowException` is thrown and no further steps are executed.
- 24 • An array instance with the given dimension lengths is allocated. If there is not enough memory available
25 to allocate the new instance, a `System.OutOfMemoryException` is thrown and no further steps are
26 executed.
- 27 • All elements of the new array instance are initialized to their default values (§12.2).
- 28 • If the array creation expression contains an array initializer, then each expression in the array initializer
29 is evaluated and assigned to its corresponding array element. The evaluations and assignments are
30 performed in the order the expressions are written in the array initializer—in other words, elements are
31 initialized in increasing index order, with the rightmost dimension increasing first. If evaluation of a
32 given expression or the subsequent assignment to the corresponding array element causes an exception,
33 then no further elements are initialized (and the remaining elements will thus have their default values).

34 An array creation expression permits instantiation of an array with elements of an array type, but the
35 elements of such an array must be manually initialized. [*Example:* For example, the statement

```
36     int[][] a = new int[100][];
```

37 creates a single-dimensional array with 100 elements of type `int[]`. The initial value of each element is
38 `null`. *end example*] It is not possible for the same array creation expression to also instantiate the sub-
39 arrays, and the statement

```
40     int[][] a = new int[100][5];    // Error
```

41 results in a compile-time error. Instantiation of the sub-arrays must instead be performed manually, as in

```
42     int[][] a = new int[100][];
43     for (int i = 0; i < 100; i++) a[i] = new int[5];
```

44 When an array of arrays has a “rectangular” shape, that is when the sub-arrays are all of the same length, it is
45 more efficient to use a multi-dimensional array. In the example above, instantiation of the array of arrays
46 creates 101 objects—one outer array and 100 sub-arrays. In contrast,

1 `int[,] = new int[100, 5];`

2 creates only a single object, a two-dimensional array, and accomplishes the allocation in a single statement.

3 14.5.10.3 Delegate creation expressions

4 A *delegate-creation-expression* is used to create a new instance of a *delegate-type*.

5 *delegate-creation-expression*:
6 `new delegate-type (expression)`

7 The argument of a delegate creation expression must be a method group (§14.1) or a value of a *delegate-type*.
8 If the argument is a method group, it identifies the method and, for an instance method, the object for
9 which to create a delegate. If the argument is a value of a *delegate-type*, it identifies a delegate instance of
10 which to create a copy.

11 The compile-time processing of a *delegate-creation-expression* of the form `new D(E)`, where D is a
12 *delegate-type* and E is an *expression*, consists of the following steps:

- 13 • If E is a method group:
- 14 ○ The set of methods identified by E must include exactly one method that is compatible (§22.1)
 - 15 with D, and this method becomes the one to which the newly created delegate refers. If no matching
 - 16 method exists, or if more than one matching method exists, a compile-time error occurs. If the
 - 17 selected method is an instance method, the instance expression associated with E determines the
 - 18 target object of the delegate.
 - 19 ○ As in a method invocation, the selected method must be compatible with the context of the method
 - 20 group: If the method is a static method, the method group must have resulted from a *simple-name* or
 - 21 a *member-access* through a type. If the method is an instance method, the method group must have
 - 22 resulted from a *simple-name* or a *member-access* through a variable or value. If the selected method
 - 23 does not match the context of the method group, a compile-time error occurs.
 - 24 ○ The result is a value of type D, namely a newly created delegate that refers to the selected method
 - 25 and target object.
- 26 • Otherwise, if E is a value of a *delegate-type*:
- 27 ○ D and E must be compatible (§22.1); otherwise, a compile-time error occurs.
 - 28 ○ The result is a value of type D, namely a newly created delegate that refers to the same invocation
 - 29 list as E.
- 30 • Otherwise, the delegate creation expression is invalid, and a compile-time error occurs.

31 The run-time processing of a *delegate-creation-expression* of the form `new D(E)`, where D is a *delegate-type*
32 and E is an *expression*, consists of the following steps:

- 33 • If E is a method group:
- 34 ○ If the method selected at compile-time is a static method, the target object of the delegate is `null`.
 - 35 Otherwise, the selected method is an instance method, and the target object of the delegate is
 - 36 determined from the instance expression associated with E:
 - 37 • The instance expression is evaluated. If this evaluation causes an exception, no further steps are
 - 38 executed.
 - 39 • If the instance expression is of a *reference-type*, the value computed by the instance expression
 - 40 becomes the target object. If the target object is `null`, a `System.NullReferenceException`
 - 41 is thrown and no further steps are executed.
 - 42 • If the instance expression is of a *value-type*, a boxing operation (§11.3.1) is performed to
 - 43 convert the value to an object, and this object becomes the target object.

- 1 ○ A new instance of the delegate type *D* is allocated. If there is not enough memory available to
- 2 allocate the new instance, a `System.OutOfMemoryException` is thrown and no further steps are
- 3 executed.
- 4 ○ The new delegate instance is initialized with a reference to the method that was determined at
- 5 compile-time and a reference to the target object computed above.
- 6 • If *E* is a value of a *delegate-type*:
- 7 ○ *E* is evaluated. If this evaluation causes an exception, no further steps are executed.
- 8 ○ If the value of *E* is `null`, a `System.NullReferenceException` is thrown and no further steps
- 9 are executed.
- 10 ○ A new instance of the delegate type *D* is allocated. If there is not enough memory available to
- 11 allocate the new instance, a `System.OutOfMemoryException` is thrown and no further steps are
- 12 executed.
- 13 ○ The new delegate instance is initialized with references to the same invocation list as the delegate
- 14 instance given by *E*.

15 The method and object to which a delegate refers are determined when the delegate is instantiated and then
 16 remain constant for the entire lifetime of the delegate. In other words, it is not possible to change the target
 17 method or object of a delegate once it has been created. [*Note*: Remember, when two delegates are
 18 combined or one is removed from another, a new delegate results; no existing delegate has its content
 19 changed. *end note*]

20 It is not possible to create a delegate that refers to a property, indexer, user-defined operator, instance
 21 constructor, destructor, or static constructor.

22 [*Example*: As described above, when a delegate is created from a method group, the formal parameter list
 23 and return type of the delegate determine which of the overloaded methods to select. In the example

```

24     delegate double DoubleFunc(double x);
25     class A
26     {
27         DoubleFunc f = new DoubleFunc(Square);
28         static float Square(float x) {
29             return x * x;
30         }
31         static double Square(double x) {
32             return x * x;
33         }
34     }

```

35 the `A.f` field is initialized with a delegate that refers to the second `Square` method because that method
 36 exactly matches the formal parameter list and return type of `DoubleFunc`. Had the second `Square` method
 37 not been present, a compile-time error would have occurred. *end example*]

38 14.5.11 The `typeof` operator

39 The `typeof` operator is used to obtain the `System.Type` object for a type.

```

40     typeof-expression:
41         typeof ( type )
42         typeof ( void )

```

43 The first form of *typeof-expression* consists of a `typeof` keyword followed by a parenthesized *type*. The
 44 result of an expression of this form is the `System.Type` object for the indicated type. There is only one
 45 `System.Type` object for any given type. [*Note*: This means that for type *T*, `typeof(T) == typeof(T)`
 46 is always true. *end note*]

1 The second form of *typeof-expression* consists of a `typeof` keyword followed by a parenthesized `void`
 2 keyword. The result of an expression of this form is the `System.Type` object that represents the absence of
 3 a type. The type object returned by `typeof(void)` is distinct from the type object returned for any type.
 4 [Note: This special type object is useful in class libraries that allow reflection onto methods in the language,
 5 where those methods wish to have a way to represent the return type of any method, including void methods,
 6 with an instance of `System.Type`. *end note*]

7 [Example: The example

```

8     using System;
9     class Test
10    {
11        static void Main() {
12            Type[] t = {
13                typeof(int),
14                typeof(System.Int32),
15                typeof(string),
16                typeof(double[]),
17                typeof(void) };
18            for (int i = 0; i < t.Length; i++) {
19                Console.WriteLine(t[i].FullName);
20            }
21        }
22    }

```

23 produces the following output:

```

24     System.Int32
25     System.Int32
26     System.String
27     System.Double[]
28     System.Void

```

29 Note that `int` and `System.Int32` are the same type. *end example*]

30 14.5.12 The checked and unchecked operators

31 The checked and unchecked operators are used to control the *overflow checking context* for integral-type
 32 arithmetic operations and conversions.

33 *checked-expression:*

34 `checked (expression)`

35 *unchecked-expression:*

36 `unchecked (expression)`

37 The `checked` operator evaluates the contained expression in a checked context, and the `unchecked`
 38 operator evaluates the contained expression in an unchecked context. A *checked-expression* or *unchecked-*
 39 *expression* corresponds exactly to a *parenthesized-expression* (§14.5.3), except that the contained expression
 40 is evaluated in the given overflow checking context.

41 The overflow checking context can also be controlled through the `checked` and `unchecked` statements
 42 (§15.11).

43 The following operations are affected by the overflow checking context established by the `checked` and
 44 `unchecked` operators and statements:

- 45 • The predefined `++` and `--` unary operators (§14.5.9 and §14.6.5), when the operand is of an integral
 46 type.
- 47 • The predefined `-` unary operator (§14.6.2), when the operand is of an integral type.
- 48 • The predefined `+`, `-`, `*`, and `/` binary operators (§14.7), when both operands are of integral types.
- 49 • Explicit numeric conversions (§13.2.1) from one integral type to another integral type.

C# LANGUAGE SPECIFICATION

1 When one of the above operations produce a result that is too large to represent in the destination type, the
2 context in which the operation is performed controls the resulting behavior:

- 3 • In a **checked** context, if the operation is a constant expression (§14.15), a compile-time error occurs.
4 Otherwise, when the operation is performed at run-time, a `System.OverflowException` is thrown.
- 5 • In an **unchecked** context, the result is truncated by discarding any high-order bits that do not fit in the
6 destination type.

7 For non-constant expressions (expressions that are evaluated at run-time) that are not enclosed by any
8 **checked** or **unchecked** operators or statements, the default overflow checking context is **unchecked**,
9 unless external factors (such as compiler switches and execution environment configuration) call for
10 **checked** evaluation.

11 For constant expressions (expressions that can be fully evaluated at compile-time), the default overflow
12 checking context is always **checked**. Unless a constant expression is explicitly placed in an **unchecked**
13 context, overflows that occur during the compile-time evaluation of the expression always cause compile-
14 time errors.

15 [*Note:* Developers may benefit if they exercise their code using checked mode (as well as unchecked mode).
16 It also seems reasonable that, unless otherwise requested, the default overflow checking context is set to
17 **checked** when debugging is enabled. *end note*]

18 [*Example:* In the example

```
19     class Test
20     {
21         static readonly int x = 1000000;
22         static readonly int y = 1000000;
23
24         static int F() {
25             return checked(x * y);    // Throws OverflowException
26         }
27
28         static int G() {
29             return unchecked(x * y); // Returns -727379968
30         }
31
32         static int H() {
33             return x * y;            // Depends on default
34         }
35     }
```

33 no compile-time errors are reported since neither of the expressions can be evaluated at compile-time. At
34 run-time, the `F` method throws a `System.OverflowException`, and the `G` method returns `-727379968`
35 (the lower 32 bits of the out-of-range result). The behavior of the `H` method depends on the default overflow
36 checking context for the compilation, but it is either the same as `F` or the same as `G`. *end example*]

37 [*Example:* In the example

```
38     class Test
39     {
40         const int x = 1000000;
41         const int y = 1000000;
42
43         static int F() {
44             return checked(x * y);    // Compile error, overflow
45         }
46
47         static int G() {
48             return unchecked(x * y); // Returns -727379968
49         }
50
51         static int H() {
52             return x * y;            // Compile error, overflow
53         }
54     }
```

1 the overflows that occur when evaluating the constant expressions in F and H cause compile-time errors to
 2 be reported because the expressions are evaluated in a `checked` context. An overflow also occurs when
 3 evaluating the constant expression in G, but since the evaluation takes place in an `unchecked` context, the
 4 overflow is not reported. *end example*]

5 The `checked` and `unchecked` operators only affect the overflow checking context for those operations that
 6 are textually contained within the “(” and “)” tokens. The operators have no effect on function members
 7 that are invoked as a result of evaluating the contained expression. [*Example*: In the example

```

8     class Test
9     {
10        static int Multiply(int x, int y) {
11            return x * y;
12        }
13        static int F() {
14            return checked(Multiply(1000000, 1000000));
15        }
16    }

```

17 the use of `checked` in F does not affect the evaluation of `x * y` in `Multiply`, so `x * y` is evaluated in
 18 the default overflow checking context. *end example*]

19 The `unchecked` operator is convenient when writing constants of the signed integral types in hexadecimal
 20 notation. [*Example*: For example:

```

21     class Test
22     {
23         public const int AllBits = unchecked((int)0xFFFFFFFF);
24         public const int HighBit = unchecked((int)0x80000000);
25     }

```

26 Both of the hexadecimal constants above are of type `uint`. Because the constants are outside the `int` range,
 27 without the `unchecked` operator, the casts to `int` would produce compile-time errors. *end example*]

28 [*Note*: The `checked` and `unchecked` operators and statements allow programmers to control certain
 29 aspects of some numeric calculations. However, the behavior of some numeric operators depends on their
 30 operands’ data types. For example, multiplying two decimals always results in an exception on overflow
 31 *even* within an explicitly `unchecked` construct. Similarly, multiplying two floats never results in an
 32 exception on overflow *even* within an explicitly `checked` construct. In addition, other operators are *never*
 33 affected by the mode of checking, whether default or explicit. As a service to programmers, it is
 34 recommended that the compiler issue a warning when there is an arithmetic expression within an explicitly
 35 `checked` or `unchecked` context (by operator or statement) that cannot possibly be affected by the specified
 36 mode of checking. Since such a warning is not required, the compiler has flexibility in determining the
 37 circumstances that merit the issuance of such warnings. *end note*]

38 14.6 Unary expressions

```

39     unary-expression:
40         primary-expression
41         + unary-expression
42         - unary-expression
43         ! unary-expression
44         ~ unary-expression
45         * unary-expression
46         & unary-expression
47     pre-increment-expression
48     pre-decrement-expression
49     cast-expression

```

1 14.6.1 Unary plus operator

2 For an operation of the form `+x`, unary operator overload resolution (§14.2.3) is applied to select a specific
3 operator implementation. The operand is converted to the parameter type of the selected operator, and the
4 type of the result is the return type of the operator. The predefined unary plus operators are:

```
5     int operator +(int x);
6     uint operator +(uint x);
7     long operator +(long x);
8     ulong operator +(ulong x);
9     float operator +(float x);
10    double operator +(double x);
11    decimal operator +(decimal x);
```

12 For each of these operators, the result is simply the value of the operand.

13 14.6.2 Unary minus operator

14 For an operation of the form `-x`, unary operator overload resolution (§14.2.3) is applied to select a specific
15 operator implementation. The operand is converted to the parameter type of the selected operator, and the
16 type of the result is the return type of the operator. The predefined negation operators are:

- 17 • Integer negation:

```
18     int operator -(int x);
19     long operator -(long x);
```

20 The result is computed by subtracting `x` from zero. In a `checked` context, if the value of `x` is the
21 maximum negative `int` or `long`, a `System.OverflowException` is thrown. In an `unchecked`
22 context, if the value of `x` is the maximum negative `int` or `long`, the result is that same value and the
23 overflow is not reported.

24 If the operand of the negation operator is of type `uint`, it is converted to type `long`, and the type of the
25 result is `long`. An exception is the rule that permits the `int` value -2^{31} to be written as a
26 decimal integer literal (§9.4.4.2).

27 If the operand of the negation operator is of type `ulong`, a compile-time error occurs. An exception is
28 the rule that permits the `long` value -2^{63} to be written as a decimal integer
29 literal (§9.4.4.2).

- 30 • Floating-point negation:

```
31     float operator -(float x);
32     double operator -(double x);
```

33 The result is the value of `x` with its sign inverted. If `x` is `NaN`, the result is also `NaN`.

- 34 • Decimal negation:

```
35     decimal operator -(decimal x);
```

36 The result is computed by subtracting `x` from zero.

37 Decimal negation is equivalent to using the unary minus operator of type `System.Decimal`.

38 14.6.3 Logical negation operator

39 For an operation of the form `!x`, unary operator overload resolution (§14.2.3) is applied to select a specific
40 operator implementation. The operand is converted to the parameter type of the selected operator, and the
41 type of the result is the return type of the operator. Only one predefined logical negation operator exists:

```
42     bool operator !(bool x);
```

43 This operator computes the logical negation of the operand: If the operand is `true`, the result is `false`. If
44 the operand is `false`, the result is `true`.

1 14.6.4 Bitwise complement operator

2 For an operation of the form `~x`, unary operator overload resolution (§14.2.3) is applied to select a specific
3 operator implementation. The operand is converted to the parameter type of the selected operator, and the
4 type of the result is the return type of the operator. The predefined bitwise complement operators are:

```
5     int operator ~(int x);
6     uint operator ~(uint x);
7     long operator ~(long x);
8     ulong operator ~(ulong x);
```

9 For each of these operators, the result of the operation is the bitwise complement of `x`.

10 Every enumeration type `E` implicitly provides the following bitwise complement operator:

```
11     E operator ~(E x);
```

12 The result of evaluating `~x`, where `x` is an expression of an enumeration type `E` with an underlying type `U`, is
13 exactly the same as evaluating `(E)(~(U)x)`.

14 14.6.5 Prefix increment and decrement operators

```
15     pre-increment-expression:  
16     ++ unary-expression
```

```
17     pre-decrement-expression:  
18     -- unary-expression
```

19 The operand of a prefix increment or decrement operation must be an expression classified as a variable, a
20 property access, or an indexer access. The result of the operation is a value of the same type as the operand.

21 If the operand of a prefix increment or decrement operation is a property or indexer access, the property or
22 indexer must have both a `get` and a `set` accessor. If this is not the case, a compile-time error occurs.

23 Unary operator overload resolution (§14.2.3) is applied to select a specific operator implementation.

24 Predefined `++` and `--` operators exist for the following types: `sbyte`, `byte`, `short`, `ushort`, `int`, `uint`,
25 `long`, `ulong`, `char`, `float`, `double`, `decimal`, and any enum type. The predefined `++` operators return
26 the value produced by adding 1 to the operand, and the predefined `--` operators return the value produced
27 by subtracting 1 from the operand.

28 The run-time processing of a prefix increment or decrement operation of the form `++x` or `--x` consists of the
29 following steps:

- 30 • If `x` is classified as a variable:
 - 31 ○ `x` is evaluated to produce the variable.
 - 32 ○ The selected operator is invoked with the value of `x` as its argument.
 - 33 ○ The value returned by the operator is stored in the location given by the evaluation of `x`.
 - 34 ○ The value returned by the operator becomes the result of the operation.
- 35 • If `x` is classified as a property or indexer access:
 - 36 ○ The instance expression (if `x` is not `static`) and the argument list (if `x` is an indexer access)
37 associated with `x` are evaluated, and the results are used in the subsequent `get` and `set` accessor
38 invocations.
 - 39 ○ The `get` accessor of `x` is invoked.
 - 40 ○ The selected operator is invoked with the value returned by the `get` accessor as its argument.
 - 41 ○ The `set` accessor of `x` is invoked with the value returned by the operator as its `value` argument.
 - 42 ○ The value returned by the operator becomes the result of the operation.

1 The ++ and -- operators also support postfix notation (§14.5.9). The result of $x++$ or $x--$ is the value of x
 2 *before* the operation, whereas the result of $++x$ or $--x$ is the value of x *after* the operation. In either case, x
 3 itself has the same value after the operation.

4 An operator ++ or operator -- implementation can be invoked using either postfix or prefix notation.
 5 It is not possible to have separate operator implementations for the two notations.

6 **14.6.6 Cast expressions**

7 A *cast-expression* is used to explicitly convert an expression to a given type.

8 *cast-expression*:
 9 (*type*) *unary-expression*

10 A *cast-expression* of the form (T)E, where T is a *type* and E is a *unary-expression*, performs an explicit
 11 conversion (§13.2) of the value of E to type T. If no explicit conversion exists from the type of E to T, a
 12 compile-time error occurs. Otherwise, the result is the value produced by the explicit conversion. The result
 13 is always classified as a value, even if E denotes a variable.

14 The grammar for a *cast-expression* leads to certain syntactic ambiguities. For example, the expression (x)-
 15 y could either be interpreted as a *cast-expression* (a cast of -y to type x) or as an *additive-expression*
 16 combined with a *parenthesized-expression* (which computes the value $x - y$).

17 To resolve *cast-expression* ambiguities, the following rule exists: A sequence of one or more *tokens* (§9.4)
 18 enclosed in parentheses is considered the start of a *cast-expression* only if at least one of the following are
 19 true:

- 20 • The sequence of tokens is correct grammar for a *type*, but not for an *expression*.
- 21 • The sequence of tokens is correct grammar for a *type*, and the token immediately following the closing
 22 parentheses is the token “~”, the token “!”, the token “(”, an *identifier* (§9.4.1), a *literal* (§9.4.4), or any
 23 *keyword* (§9.4.3) except `as` and `is`.

24 [*Note*: The above rule means that only if the construct is unambiguously a *cast-expression* is it considered a
 25 *cast-expression*. *end note*]

26 The term “correct grammar” above means only that the sequence of tokens must conform to the particular
 27 grammatical production. It specifically does not consider the actual meaning of any constituent identifiers.
 28 For example, if x and y are identifiers, then $x.y$ is correct grammar for a *type*, even if $x.y$ doesn’t actually
 29 denote a *type*.

30 [*Note*: From the disambiguation rule, it follows that, if x and y are identifiers, $(x)y$, $(x)(y)$, and $(x)(-y)$
 31 are *cast-expressions*, but $(x)-y$ is not, even if x identifies a *type*. However, if x is a *keyword* that identifies
 32 a predefined *type* (such as `int`), then all four forms are *cast-expressions* (because such a *keyword* could not
 33 possibly be an *expression* by itself). *end note*]

34 **14.7 Arithmetic operators**

35 The *, /, %, +, and - operators are called the arithmetic operators.

36 *multiplicative-expression*:
 37 *unary-expression*
 38 *multiplicative-expression* * *unary-expression*
 39 *multiplicative-expression* / *unary-expression*
 40 *multiplicative-expression* % *unary-expression*

41 *additive-expression*:
 42 *multiplicative-expression*
 43 *additive-expression* + *multiplicative-expression*
 44 *additive-expression* - *multiplicative-expression*

14.7.1 Multiplication operator

For an operation of the form $x * y$, binary operator overload resolution (§14.2.4) is applied to select a specific operator implementation. The operands are converted to the parameter types of the selected operator, and the type of the result is the return type of the operator.

The predefined multiplication operators are listed below. The operators all compute the product of x and y .

- Integer multiplication:

```
int operator *(int x, int y);
uint operator *(uint x, uint y);
long operator *(long x, long y);
ulong operator *(ulong x, ulong y);
```

In a `checked` context, if the product is outside the range of the result type, a `System.OverflowException` is thrown. In an `unchecked` context, overflows are not reported and any significant high-order bits outside the range of the result type are discarded.

- Floating-point multiplication:

```
float operator *(float x, float y);
double operator *(double x, double y);
```

The product is computed according to the rules of IEEE 754 arithmetic. The following table lists the results of all possible combinations of nonzero finite values, zeros, infinities, and NaN's. In the table, x and y are positive finite values. z is the result of $x * y$. If the result is too large for the destination type, z is infinity. If the result is too small for the destination type, z is zero.

	+y	-y	+0	-0	+∞	-∞	NaN
+x	+z	-z	+0	-0	+∞	-∞	NaN
-x	-z	+z	-0	+0	-∞	+∞	NaN
+0	+0	-0	+0	-0	NaN	NaN	NaN
-0	-0	+0	-0	+0	NaN	NaN	NaN
+∞	+∞	-∞	NaN	NaN	+∞	-∞	NaN
-∞	-∞	+∞	NaN	NaN	-∞	+∞	NaN
NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN

- Decimal multiplication:

```
decimal operator *(decimal x, decimal y);
```

If the resulting value is too large to represent in the `decimal` format, a `System.OverflowException` is thrown. If the result value is too small to represent in the `decimal` format, the result is zero. The scale of the result, before any rounding, is the sum of the scales of the two operands.

Decimal multiplication is equivalent to using the multiplication operator of type `System.Decimal`.

14.7.2 Division operator

For an operation of the form x / y , binary operator overload resolution (§14.2.4) is applied to select a specific operator implementation. The operands are converted to the parameter types of the selected operator, and the type of the result is the return type of the operator.

The predefined division operators are listed below. The operators all compute the quotient of x and y .

- Integer division:

```
int operator /(int x, int y);
uint operator /(uint x, uint y);
long operator /(long x, long y);
ulong operator /(ulong x, ulong y);
```

1 If the value of the right operand is zero, a `System.DivideByZeroException` is thrown.

2 The division rounds the result towards zero, and the absolute value of the result is the largest possible
3 integer that is less than the absolute value of the quotient of the two operands. The result is zero or
4 positive when the two operands have the same sign and zero or negative when the two operands have
5 opposite signs.

6 If the left operand is the maximum negative `int` or `long` value and the right operand is `-1`, an overflow
7 occurs. In a checked context, this causes a `System.OverflowException` to be thrown. In an
8 unchecked context, the overflow is not reported and the result is instead the value of the left operand.

9 • Floating-point division:

10 `float operator /(float x, float y);`
11 `double operator /(double x, double y);`

12 The quotient is computed according to the rules of IEEE 754 arithmetic. The following table lists the
13 results of all possible combinations of nonzero finite values, zeros, infinities, and NaN's. In the table, `x`
14 and `y` are positive finite values. `z` is the result of `x / y`. If the result is too large for the destination type,
15 `z` is infinity. If the result is too small for the destination type, `z` is zero.

16

	<code>+y</code>	<code>-y</code>	<code>+0</code>	<code>-0</code>	<code>+∞</code>	<code>-∞</code>	NaN
<code>+x</code>	<code>+z</code>	<code>-z</code>	<code>+∞</code>	<code>-∞</code>	<code>+0</code>	<code>-0</code>	NaN
<code>-x</code>	<code>-z</code>	<code>+z</code>	<code>-∞</code>	<code>+∞</code>	<code>-0</code>	<code>+0</code>	NaN
<code>+0</code>	<code>+0</code>	<code>-0</code>	NaN	NaN	<code>+0</code>	<code>-0</code>	NaN
<code>-0</code>	<code>-0</code>	<code>+0</code>	NaN	NaN	<code>-0</code>	<code>+0</code>	NaN
<code>+∞</code>	<code>+∞</code>	<code>-∞</code>	<code>+∞</code>	<code>-∞</code>	NaN	NaN	NaN
<code>-∞</code>	<code>-∞</code>	<code>+∞</code>	<code>-∞</code>	<code>+∞</code>	NaN	NaN	NaN
NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN

17

18 • Decimal division:

19 `decimal operator /(decimal x, decimal y);`

20 If the value of the right operand is zero, a `System.DivideByZeroException` is thrown. If the
21 resulting value is too large to represent in the `decimal` format, a `System.OverflowException` is
22 thrown. If the result value is too small to represent in the `decimal` format, the result is zero. The scale
23 of the result, before any rounding, is the smallest scale that will preserve a result equal to the exact
24 result.

25 Decimal division is equivalent to using the division operator of type `System.Decimal`.

26 14.7.3 Remainder operator

27 For an operation of the form `x % y`, binary operator overload resolution (§14.2.4) is applied to select a
28 specific operator implementation. The operands are converted to the parameter types of the selected
29 operator, and the type of the result is the return type of the operator.

30 The predefined remainder operators are listed below. The operators all compute the remainder of the
31 division between `x` and `y`.

32 • Integer remainder:

33 `int operator %(int x, int y);`
34 `uint operator %(uint x, uint y);`
35 `long operator %(long x, long y);`
36 `ulong operator %(ulong x, ulong y);`

37 The result of `x % y` is the value produced by `x - (x / y) * y`. If `y` is zero, a
38 `System.DivideByZeroException` is thrown. The remainder operator never causes an overflow.

- 1 • Floating-point remainder:

2 float operator %(float x, float y);
3 double operator %(double x, double y);

4 The following table lists the results of all possible combinations of nonzero finite values, zeros,
5 infinities, and NaN's. In the table, x and y are positive finite values. z is the result of $x \% y$ and is
6 computed as $x - n * y$, where n is the largest possible integer that is less than or equal to x / y . This
7 method of computing the remainder is analogous to that used for integer operands, but differs from the
8 IEEE 754 definition (in which n is the integer closest to x / y).

9

	+y	-y	+0	-0	+∞	-∞	NaN
+x	+z	+z	NaN	NaN	x	x	NaN
-x	-z	-z	NaN	NaN	-x	-x	NaN
+0	+0	+0	NaN	NaN	+0	+0	NaN
-0	-0	-0	NaN	NaN	-0	-0	NaN
+∞	NaN	NaN	NaN	NaN	NaN	NaN	NaN
-∞	NaN	NaN	NaN	NaN	NaN	NaN	NaN
NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN

10

- 11 • Decimal remainder:

12 decimal operator %(decimal x, decimal y);

13 If the value of the right operand is zero, a `System.DivideByZeroException` is thrown. If the
14 resulting value is too large to represent in the `decimal` format, a `System.OverflowException` is
15 thrown. If the result value is too small to represent in the `decimal` format, the result is zero. The scale
16 of the result, before any rounding, is the same as the scale of y, and the sign of the result, if non-zero, is
17 the same as that of x.

18 Decimal remainder is equivalent to using the remainder operator of type `System.Decimal`.

19 14.7.4 Addition operator

20 For an operation of the form $x + y$, binary operator overload resolution (§14.2.4) is applied to select a
21 specific operator implementation. The operands are converted to the parameter types of the selected
22 operator, and the type of the result is the return type of the operator.

23 The predefined addition operators are listed below. For numeric and enumeration types, the predefined
24 addition operators compute the sum of the two operands. When one or both operands are of type `string`,
25 the predefined addition operators concatenate the string representation of the operands.

- 26 • Integer addition:

27 int operator +(int x, int y);
28 uint operator +(uint x, uint y);
29 long operator +(long x, long y);
30 ulong operator +(ulong x, ulong y);

31 In a `checked` context, if the sum is outside the range of the result type, a
32 `System.OverflowException` is thrown. In an `unchecked` context, overflows are not reported and
33 any significant high-order bits outside the range of the result type are discarded.

- 34 • Floating-point addition:

35 float operator +(float x, float y);
36 double operator +(double x, double y);

37 The sum is computed according to the rules of IEEE 754 arithmetic. The following table lists the results
38 of all possible combinations of nonzero finite values, zeros, infinities, and NaN's. In the table, x and y
39 are nonzero finite values, and z is the result of $x + y$. If x and y have the same magnitude but opposite

signs, z is positive zero. If $x + y$ is too large to represent in the destination type, z is an infinity with the same sign as $x + y$. If $x + y$ is too small to represent in the destination type, z is a zero with the same sign as $x + y$.

	y	$+0$	-0	$+\infty$	$-\infty$	NaN
x	z	x	x	$+\infty$	$-\infty$	NaN
$+0$	y	$+0$	$+0$	$+\infty$	$-\infty$	NaN
-0	y	$+0$	-0	$+\infty$	$-\infty$	NaN
$+\infty$	$+\infty$	$+\infty$	$+\infty$	$+\infty$	NaN	NaN
$-\infty$	$-\infty$	$-\infty$	$-\infty$	NaN	$-\infty$	NaN
NaN	NaN	NaN	NaN	NaN	NaN	NaN

- Decimal addition:

```
decimal operator +(decimal x, decimal y);
```

If the resulting value is too large to represent in the `decimal` format, a `System.OverflowException` is thrown. The scale of the result, before any rounding, is the larger of the scales of the two operands.

Decimal addition is equivalent to using the addition operator of type `System.Decimal`.

- Enumeration addition. Every enumeration type implicitly provides the following predefined operators, where E is the enum type, and U is the underlying type of E :

```
E operator +(E x, U y);
E operator +(U x, E y);
```

The operators are evaluated exactly as $(E)((U)x + (U)y)$.

- String concatenation:

```
string operator +(string x, string y);
string operator +(string x, object y);
string operator +(object x, string y);
```

The binary `+` operator performs string concatenation when one or both operands are of type `string`. If an operand of string concatenation is `null`, an empty string is substituted. Otherwise, any non-string argument is converted to its string representation by invoking the virtual `ToString` method inherited from type `object`. If `ToString` returns `null`, an empty string is substituted. [Example:

```
using System;
class Test
{
    static void Main() {
        string s = null;
        Console.WriteLine("s = >" + s + "<"); // displays s = ><
        int i = 1;
        Console.WriteLine("i = " + i); // displays i = 1
        float f = 1.2300E+15F;
        Console.WriteLine("f = " + f); // displays f = 1.23E+15
        decimal d = 2.900m;
        Console.WriteLine("d = " + d); // displays d = 2.900
    }
}
```

end example]

The result of the string concatenation operator is a string that consists of the characters of the left operand followed by the characters of the right operand. The string concatenation operator never returns a `null` value. A `System.OutOfMemoryException` may be thrown if there is not enough memory available to allocate the resulting string.

- 1 • Delegate combination. Every delegate type implicitly provides the following predefined operator, where
2 D is the delegate type:

3 `D operator +(D x, D y);`

4 The binary `+` operator performs delegate combination when both operands are of some delegate type D.
5 (If the operands have different delegate types, a compile-time error occurs.) If the first operand is `null`,
6 the result of the operation is the value of the second operand (even if that is also `null`). Otherwise, if the
7 second operand is `null`, then the result of the operation is the value of the first operand. Otherwise, the
8 result of the operation is a new delegate instance that, when invoked, invokes the first operand and then
9 invokes the second operand. [Note: For examples of delegate combination, see §14.7.5 and §22.3. Since
10 `System.Delegate` is not a delegate type, operator `+` is not defined for it. *end note*]

11 14.7.5 Subtraction operator

12 For an operation of the form `x - y`, binary operator overload resolution (§14.2.4) is applied to select a
13 specific operator implementation. The operands are converted to the parameter types of the selected
14 operator, and the type of the result is the return type of the operator.

15 The predefined subtraction operators are listed below. The operators all subtract `y` from `x`.

- 16 • Integer subtraction:

17 `int operator -(int x, int y);`
18 `uint operator -(uint x, uint y);`
19 `long operator -(long x, long y);`
20 `ulong operator -(ulong x, ulong y);`

21 In a `checked` context, if the difference is outside the range of the result type, a
22 `System.OverflowException` is thrown. In an `unchecked` context, overflows are not reported and
23 any significant high-order bits outside the range of the result type are discarded.

- 24 • Floating-point subtraction:

25 `float operator -(float x, float y);`
26 `double operator -(double x, double y);`

27 The difference is computed according to the rules of IEEE 754 arithmetic. The following table lists the
28 results of all possible combinations of nonzero finite values, zeros, infinities, and NaNs. In the table, `x`
29 and `y` are nonzero finite values, and `z` is the result of `x - y`. If `x` and `y` are equal, `z` is positive zero. If
30 `x - y` is too large to represent in the destination type, `z` is an infinity with the same sign as `x - y`. If
31 `x - y` is too small to represent in the destination type, `z` is a zero with the same sign as `x - y`.

	<code>y</code>	<code>+0</code>	<code>-0</code>	<code>+∞</code>	<code>-∞</code>	NaN
<code>x</code>	<code>z</code>	<code>x</code>	<code>x</code>	<code>-∞</code>	<code>+∞</code>	NaN
<code>+0</code>	<code>-y</code>	<code>+0</code>	<code>+0</code>	<code>-∞</code>	<code>+∞</code>	NaN
<code>-0</code>	<code>-y</code>	<code>-0</code>	<code>+0</code>	<code>-∞</code>	<code>+∞</code>	NaN
<code>+∞</code>	<code>+∞</code>	<code>+∞</code>	<code>+∞</code>	NaN	<code>+∞</code>	NaN
<code>-∞</code>	<code>-∞</code>	<code>-∞</code>	<code>-∞</code>	<code>-∞</code>	NaN	NaN
NaN	NaN	NaN	NaN	NaN	NaN	NaN

- 33
34 • Decimal subtraction:

35 `decimal operator -(decimal x, decimal y);`

36 If the resulting value is too large to represent in the `decimal` format, a `System.OverflowException`
37 is thrown. The scale of the result, before any rounding, is the larger of the scales of the two operands.

38 Decimal subtraction is equivalent to using the subtraction operator of type `System.Decimal`.

- 1 • Enumeration subtraction. Every enumeration type implicitly provides the following predefined operator,
2 where E is the enum type, and U is the underlying type of E:

3 U operator -(E x, E y);

4 This operator is evaluated exactly as $(U)((U)x - (U)y)$. In other words, the operator computes the
5 difference between the ordinal values of x and y, and the type of the result is the underlying type of the
6 enumeration.

7 E operator -(E x, U y);

8 This operator is evaluated exactly as $(E)((U)x - y)$. In other words, the operator subtracts a value
9 from the underlying type of the enumeration, yielding a value of the enumeration.

- 10 • Delegate removal. Every delegate type implicitly provides the following predefined operator, where D is
11 the delegate type:

12 D operator -(D x, D y);

13 The binary - operator performs delegate removal when both operands are of some delegate type D. (If
14 the operands have different delegate types, a compile-time error occurs.) If the first operand is null, the
15 result of the operation is null. Otherwise, if the second operand is null, then the result of the operation
16 is the value of the first operand. Otherwise, both operands represent invocation lists (§22.1) having one
17 or more entries, and the result is a new invocation list consisting of the first operand's list with the
18 second operand's entries removed from it, provided the second operand's list is a proper contiguous
19 subset of the first's. (For determining subset equality, corresponding entries are compared as for the
20 delegate equality operator (§14.9.8).) Otherwise, the result is the value of the left operand. Neither of the
21 operands' lists is changed in the process. If the second operand's list matches multiple subsets of
22 contiguous entries in the first operand's list, the right-most matching subset of contiguous entries is
23 removed. If removal results in an empty list, the result is null. [Example: For example:

```
24 using System;
25 delegate void D(int x);
26 class Test
27 {
28     public static void M1(int i) { /* ... */ }
29     public static void M2(int i) { /* ... */ }
30 }
31 class Demo
32 {
33     static void Main() {
34         D cd1 = new D(Test.M1);
35         D cd2 = new D(Test.M2);
36         D cd3 = cd1 + cd2 + cd2 + cd1; // M1 + M2 + M2 + M1
37         cd3 -= cd1; // => M1 + M2 + M2
38         cd3 = cd1 + cd2 + cd2 + cd1; // M1 + M2 + M2 + M1
39         cd3 -= cd1 + cd2; // => M2 + M1
40         cd3 = cd1 + cd2 + cd2 + cd1; // M1 + M2 + M2 + M1
41         cd3 -= cd2 + cd2; // => M1 + M1
42         cd3 = cd1 + cd2 + cd2 + cd1; // M1 + M2 + M2 + M1
43         cd3 -= cd2 + cd1; // => M1 + M2
44         cd3 = cd1 + cd2 + cd2 + cd1; // M1 + M2 + M2 + M1
45         cd3 -= cd1 + cd1; // => M1 + M2 + M2 + M1
46     }
47 }
```

48 *end example]*

49 14.8 Shift operators

50 The << and >> operators are used to perform bit shifting operations.


```

1      shift-expression:
2      additive-expression
3      shift-expression << additive-expression
4      shift-expression >> additive-expression

```

5 For an operation of the form `x << count` or `x >> count`, binary operator overload resolution (§14.2.4) is applied to select a specific operator implementation. The operands are converted to the parameter types of the selected operator, and the type of the result is the return type of the operator.

8 When declaring an overloaded shift operator, the type of the first operand must always be the class or struct containing the operator declaration, and the type of the second operand must always be `int`.

10 The predefined shift operators are listed below.

- 11 • Shift left:

```

12      int operator <<(int x, int count);
13      uint operator <<(uint x, int count);
14      long operator <<(long x, int count);
15      ulong operator <<(ulong x, int count);

```

16 The `<<` operator shifts `x` left by a number of bits computed as described below.

17 The high-order bits outside the range of the result type of `x` are discarded, the remaining bits are shifted left, and the low-order empty bit positions are set to zero.

- 19 • Shift right:

```

20      int operator >>(int x, int count);
21      uint operator >>(uint x, int count);
22      long operator >>(long x, int count);
23      ulong operator >>(ulong x, int count);

```

24 The `>>` operator shifts `x` right by a number of bits computed as described below.

25 When `x` is of type `int` or `long`, the low-order bits of `x` are discarded, the remaining bits are shifted right, and the high-order empty bit positions are set to zero if `x` is non-negative and set to one if `x` is negative.

28 When `x` is of type `uint` or `ulong`, the low-order bits of `x` are discarded, the remaining bits are shifted right, and the high-order empty bit positions are set to zero.

30 For the predefined operators, the number of bits to shift is computed as follows:

- 31 • When the type of `x` is `int` or `uint`, the shift count is given by the low-order five bits of `count`. In other words, the shift count is computed from `count & 0x1F`.
- 33 • When the type of `x` is `long` or `ulong`, the shift count is given by the low-order six bits of `count`. In other words, the shift count is computed from `count & 0x3F`.

35 If the resulting shift count is zero, the shift operators simply return the value of `x`.

36 Shift operations never cause overflows and produce the same results in `checked` and `unchecked` contexts.

37 When the left operand of the `>>` operator is of a signed integral type, the operator performs an *arithmetic* shift right wherein the value of the most significant bit (the sign bit) of the operand is propagated to the high-order empty bit positions. When the left operand of the `>>` operator is of an unsigned integral type, the operator performs a *logical* shift right wherein high-order empty bit positions are always set to zero. To perform the opposite operation of that inferred from the operand type, explicit casts can be used. For example, if `x` is a variable of type `int`, the operation `unchecked((int)((uint)x >> y))` performs a logical shift right of `x`.

44 14.9 Relational and type-testing operators

45 The `==`, `!=`, `<`, `>`, `<=`, `>=`, `is` and `as` operators are called the relational and type-testing operators.

```

1      relational-expression:
2          shift-expression
3          relational-expression < shift-expression
4          relational-expression > shift-expression
5          relational-expression <= shift-expression
6          relational-expression >= shift-expression
7          relational-expression is type
8          relational-expression as type

```

```

9      equality-expression:
10         relational-expression
11         equality-expression == relational-expression
12         equality-expression != relational-expression

```

13 The `is` operator is described in §14.9.9 and the `as` operator is described in §14.9.10.

14 The `==`, `!=`, `<`, `>`, `<=` and `>=` operators are **comparison operators**. For an operation of the form `x op y`,
15 where `op` is a comparison operator, overload resolution (§14.2.4) is applied to select a specific operator
16 implementation. The operands are converted to the parameter types of the selected operator, and the type of
17 the result is the return type of the operator.

18 The predefined comparison operators are described in the following sections. All predefined comparison
19 operators return a result of type `bool`, as described in the following table.

20

Operation	Result
<code>x == y</code>	true if x is equal to y, false otherwise
<code>x != y</code>	true if x is not equal to y, false otherwise
<code>x < y</code>	true if x is less than y, false otherwise
<code>x > y</code>	true if x is greater than y, false otherwise
<code>x <= y</code>	true if x is less than or equal to y, false otherwise
<code>x >= y</code>	true if x is greater than or equal to y, false otherwise

21

22 14.9.1 Integer comparison operators

23 The predefined integer comparison operators are:

```

24     bool operator ==(int x, int y);
25     bool operator ==(uint x, uint y);
26     bool operator ==(long x, long y);
27     bool operator ==(ulong x, ulong y);
28
29     bool operator !=(int x, int y);
30     bool operator !=(uint x, uint y);
31     bool operator !=(long x, long y);
32     bool operator !=(ulong x, ulong y);
33
34     bool operator <(int x, int y);
35     bool operator <(uint x, uint y);
36     bool operator <(long x, long y);
37     bool operator <(ulong x, ulong y);
38
39     bool operator >(int x, int y);
40     bool operator >(uint x, uint y);
41     bool operator >(long x, long y);
42     bool operator >(ulong x, ulong y);
43
44     bool operator <=(int x, int y);
45     bool operator <=(uint x, uint y);
46     bool operator <=(long x, long y);
47     bool operator <=(ulong x, ulong y);

```

```

1      bool operator >=(int x, int y);
2      bool operator >=(uint x, uint y);
3      bool operator >=(long x, long y);
4      bool operator >=(ulong x, ulong y);

```

5 Each of these operators compares the numeric values of the two integer operands and returns a `bool` value
6 that indicates whether the particular relation is `true` or `false`.

7 14.9.2 Floating-point comparison operators

8 The predefined floating-point comparison operators are:

```

9      bool operator ==(float x, float y);
10     bool operator ==(double x, double y);
11     bool operator !=(float x, float y);
12     bool operator !=(double x, double y);
13     bool operator <(float x, float y);
14     bool operator <(double x, double y);
15     bool operator >(float x, float y);
16     bool operator >(double x, double y);
17     bool operator <=(float x, float y);
18     bool operator <=(double x, double y);
19     bool operator >=(float x, float y);
20     bool operator >=(double x, double y);

```

21 The operators compare the operands according to the rules of the IEEE 754 standard:

- 22 • If either operand is NaN, the result is `false` for all operators except `!=`, for which the result is `true`.
23 For any two operands, `x != y` always produces the same result as `!(x == y)`. However, when one or
24 both operands are NaN, the `<`, `>`, `<=`, and `>=` operators *do not* produce the same results as the logical
25 negation of the opposite operator. [*Example:* For example, if either of `x` and `y` is NaN, then `x < y` is
26 `false`, but `!(x >= y)` is `true`. *end example*]

- 27 • When neither operand is NaN, the operators compare the values of the two floating-point operands with
28 respect to the ordering

```
29     -∞ < -max < ... < -min < -0.0 == +0.0 < +min < ... < +max < +∞
```

30 where `min` and `max` are the smallest and largest positive finite values that can be represented in the given
31 floating-point format. Notable effects of this ordering are:

- 32 ○ Negative and positive zeros are considered equal.
- 33 ○ A negative infinity is considered less than all other values, but equal to another negative infinity.
- 34 ○ A positive infinity is considered greater than all other values, but equal to another positive infinity.

35 14.9.3 Decimal comparison operators

36 The predefined decimal comparison operators are:

```

37     bool operator ==(decimal x, decimal y);
38     bool operator !=(decimal x, decimal y);
39     bool operator <(decimal x, decimal y);
40     bool operator >(decimal x, decimal y);
41     bool operator <=(decimal x, decimal y);
42     bool operator >=(decimal x, decimal y);

```

43 Each of these operators compares the numeric values of the two decimal operands and returns a `bool`
44 value that indicates whether the particular relation is `true` or `false`. Each decimal comparison is
45 equivalent to using the corresponding relational or equality operator of type `System.Decimal`.

46 14.9.4 Boolean equality operators

47 The predefined boolean equality operators are:

```

1      bool operator ==(bool x, bool y);
2      bool operator !=(bool x, bool y);

```

3 The result of `==` is `true` if both `x` and `y` are `true` or if both `x` and `y` are `false`. Otherwise, the result is
4 `false`.

5 The result of `!=` is `false` if both `x` and `y` are `true` or if both `x` and `y` are `false`. Otherwise, the result is
6 `true`. When the operands are of type `bool`, the `!=` operator produces the same result as the `^` operator.

7 **14.9.5 Enumeration comparison operators**

8 Every enumeration type implicitly provides the following predefined comparison operators:

```

9      bool operator ==(E x, E y);
10     bool operator !=(E x, E y);
11     bool operator <(E x, E y);
12     bool operator >(E x, E y);
13     bool operator <=(E x, E y);
14     bool operator >=(E x, E y);

```

15 The result of evaluating `x op y`, where `x` and `y` are expressions of an enumeration type `E` with an underlying
16 type `U`, and `op` is one of the comparison operators, is exactly the same as evaluating `((U)x) op ((U)y)`. In
17 other words, the enumeration type comparison operators simply compare the underlying integral values of
18 the two operands.

19 **14.9.6 Reference type equality operators**

20 The predefined reference type equality operators are:

```

21     bool operator ==(object x, object y);
22     bool operator !=(object x, object y);

```

23 The operators return the result of comparing the two references for equality or non-equality.

24 Since the predefined reference type equality operators accept operands of type `object`, they apply to all
25 types that do not declare applicable `operator ==` and `operator !=` members. Conversely, any
26 applicable user-defined equality operators effectively hide the predefined reference type equality operators.

27 The predefined reference type equality operators require the operands to be *reference-type* values or the
28 value `null`; furthermore, they require that a standard implicit conversion (§13.3.1) exists from the type of
29 either operand to the type of the other operand. Unless both of these conditions are true, a compile-time error
30 occurs. [*Note:* Notable implications of these rules are:

- 31 • It is a compile-time error to use the predefined reference type equality operators to compare two
32 references that are known to be different at compile-time. For example, if the compile-time types of the
33 operands are two class types `A` and `B`, and if neither `A` nor `B` derives from the other, then it would be
34 impossible for the two operands to reference the same object. Thus, the operation is considered a
35 compile-time error.
- 36 • The predefined reference type equality operators do not permit value type operands to be compared.
37 Therefore, unless a struct type declares its own equality operators, it is not possible to compare values of
38 that struct type.
- 39 • The predefined reference type equality operators never cause boxing operations to occur for their
40 operands. It would be meaningless to perform such boxing operations, since references to the newly
41 allocated boxed instances would necessarily differ from all other references.

42 *end note*]

43 For an operation of the form `x == y` or `x != y`, if any applicable `operator ==` or `operator !=` exists,
44 the operator overload resolution (§14.2.4) rules will select that operator instead of the predefined reference
45 type equality operator. However, it is always possible to select the predefined reference type equality
46 operator by explicitly casting one or both of the operands to type `object`. [*Example:* The example

```

1      Using System;
2      class Test
3      {
4          static void Main() {
5              string s = "Test";
6              string t = string.Copy(s);
7              Console.WriteLine(s == t);
8              Console.WriteLine((object)s == t);
9              Console.WriteLine(s == (object)t);
10             Console.WriteLine((object)s == (object)t);
11         }
12     }

```

13 produces the output

```

14     True
15     False
16     False
17     False

```

18 The `s` and `t` variables refer to two distinct `string` instances containing the same characters. The first
19 comparison outputs `True` because the predefined string equality operator (§14.9.7) is selected when both
20 operands are of type `string`. The remaining comparisons all output `False` because the predefined
21 reference type equality operator is selected when one or both of the operands are of type `object`.

22 Note that the above technique is not meaningful for value types. The example

```

23     class Test
24     {
25         static void Main() {
26             int i = 123;
27             int j = 123;
28             System.Console.WriteLine((object)i == (object)j);
29         }
30     }

```

31 outputs `False` because the casts create references to two separate instances of boxed `int` values. *end*
32 *example*]

33 14.9.7 String equality operators

34 The predefined string equality operators are: :

```

35     bool operator ==(string x, string y);
36     bool operator !=(string x, string y);

```

37 Two `string` values are considered equal when one of the following is true:

- 38 • Both values are `null`.
- 39 • Both values are non-null references to string instances that have identical lengths and identical
40 characters in each character position.

41 The string equality operators compare string *values* rather than string *references*. When two separate string
42 instances contain the exact same sequence of characters, the values of the strings are equal, but the
43 references are different. [*Note*: As described in §14.9.6, the reference type equality operators can be used to
44 compare string references instead of string values. *end note*]

45 14.9.8 Delegate equality operators

46 Every delegate type implicitly provides the following predefined comparison operators: :

```

47     bool operator ==(System.Delegate x, System.Delegate y);
48     bool operator !=(System.Delegate x, System.Delegate y);

```

49 Two delegate instances are considered equal as follows:

- 50 • If either of the delegate instances is `null`, they are equal if and only if both are `null`.

- 1 • If either of the delegate instances has an invocation list (§22.1) containing one entry, they are equal if
2 and only if the other also has an invocation list containing one entry, and either:
- 3 • Both refer to the same static method, or
- 4 • Both refer to the same non-static method on the same target object.
- 5 • If either of the delegate instances has an invocation list containing two or more entries, those instances
6 are equal if and only if their invocation lists are the same length, and each entry in one's invocation list
7 is equal to the corresponding entry, in order, in the other's invocation list.
- 8 Note that delegates of different types can be considered equal by the above definition, as long as they have
9 the same return type and parameter types.

10 14.9.9 The `is` operator

11 The `is` operator is used to dynamically check if the run-time type of an object is compatible with a given
12 type. The result of the operation `e is T`, where `e` is an expression and `T` is a type, is a boolean value
13 indicating whether `e` can successfully be converted to type `T` by a reference conversion, a boxing conversion,
14 or an unboxing conversion. The operation is evaluated as follows:

- 15 • If the compile-time type of `e` is the same as `T`, or if an implicit reference conversion (§13.1.4) or boxing
16 conversion (§13.1.5) exists from the compile-time type of `e` to `T`:
- 17 ○ If `e` is of a reference type, the result of the operation is equivalent to evaluating `e != null`.
- 18 ○ If `e` is of a value type, the result of the operation is `true`.
- 19 • Otherwise, if an explicit reference conversion (§13.2.3) or unboxing conversion (§13.2.4) exists from
20 the compile-time type of `e` to `T`, a dynamic type check is performed:
- 21 ○ If the value of `e` is `null`, the result is `false`.
- 22 ○ Otherwise, let `R` be the run-time type of the instance referenced by `e`. If `R` and `T` are the same type, if
23 `R` is a reference type and an implicit reference conversion from `R` to `T` exists, or if `R` is a value type
24 and `T` is an interface type that is implemented by `R`, the result is `true`.
- 25 ○ Otherwise, the result is `false`.
- 26 • Otherwise, no reference or boxing conversion of `e` to type `T` is possible, and the result of the operation is
27 `false`.

28 Note that the `is` operator only considers reference conversions, boxing conversions, and unboxing
29 conversions. Other conversions, such as user defined conversions, are not considered by the `is` operator.

30 14.9.10 The `as` operator

31 The `as` operator is used to explicitly convert a value to a given reference type using a reference conversion
32 or a boxing conversion. Unlike a cast expression (§14.6.6), the `as` operator never throws an exception.
33 Instead, if the indicated conversion is not possible, the resulting value is `null`.

34 In an operation of the form `e as T`, `e` must be an expression and `T` must be a reference type. The type of the
35 result is `T`, and the result is always classified as a value. The operation is evaluated as follows:

- 36 • If the compile-time type of `e` is the same as `T`, the result is simply the value of `e`.
- 37 • Otherwise, if an implicit reference conversion (§13.1.4) or boxing conversion (§13.1.5) exists from the
38 compile-time type of `e` to `T`, this conversion is performed and becomes the result of the operation.
- 39 • Otherwise, if an explicit reference conversion (§13.2.3) exists from the compile-time type of `e` to `T`, a
40 dynamic type check is performed:
- 41 ○ If the value of `e` is `null`, the result is the value `null` with the compile-time type `T`.

- 1 o Otherwise, let R be the run-time type of the instance referenced by e. If R and T are the same type, if
- 2 R is a reference type and an implicit reference conversion from R to T exists, or if R is a value type
- 3 and T is an interface type that is implemented by R, the result is the reference given by e with the
- 4 compile-time type T.
- 5 o Otherwise, the result is the value `null` with the compile-time type T.
- 6 • Otherwise, the indicated conversion is never possible, and a compile-time error occurs.

7 Note that the `as` operator only performs reference conversions and boxing conversions. Other conversions,
8 such as user defined conversions, are not possible with the `as` operator and should instead be performed
9 using cast expressions.

10 14.10 Logical operators

11 The `&`, `^`, and `|` operators are called the logical operators.

```
12     and-expression:
13         equality-expression
14         and-expression & equality-expression
15
16     exclusive-or-expression:
17         and-expression
18         exclusive-or-expression ^ and-expression
19
20     inclusive-or-expression:
21         exclusive-or-expression
22         inclusive-or-expression | exclusive-or-expression
```

21 For an operation of the form `x op y`, where `op` is one of the logical operators, overload resolution (§14.2.4) is
22 applied to select a specific operator implementation. The operands are converted to the parameter types of
23 the selected operator, and the type of the result is the return type of the operator.

24 The predefined logical operators are described in the following sections.

25 14.10.1 Integer logical operators

26 The predefined integer logical operators are:

```
27     int operator &(int x, int y);
28     uint operator &(uint x, uint y);
29     long operator &(long x, long y);
30     ulong operator &(ulong x, ulong y);
31
32     int operator |(int x, int y);
33     uint operator |(uint x, uint y);
34     long operator |(long x, long y);
35     ulong operator |(ulong x, ulong y);
36
37     int operator ^(int x, int y);
38     uint operator ^(uint x, uint y);
39     long operator ^(long x, long y);
40     ulong operator ^(ulong x, ulong y);
```

39 The `&` operator computes the bitwise logical AND of the two operands, the `|` operator computes the bitwise
40 logical OR of the two operands, and the `^` operator computes the bitwise logical exclusive OR of the two
41 operands. No overflows are possible from these operations.

42 14.10.2 Enumeration logical operators

43 Every enumeration type E implicitly provides the following predefined logical operators:

```
44     E operator &(E x, E y);
45     E operator |(E x, E y);
46     E operator ^(E x, E y);
```

1 The result of evaluating $x \text{ op } y$, where x and y are expressions of an enumeration type E with an underlying
2 type U , and op is one of the logical operators, is exactly the same as evaluating $(E)((U)x \text{ op } (U)y)$. In other
3 words, the enumeration type logical operators simply perform the logical operation on the underlying type of
4 the two operands.

5 **14.10.3 Boolean logical operators**

6 The predefined boolean logical operators are:

```
7     bool operator &(bool x, bool y);  
8     bool operator |(bool x, bool y);  
9     bool operator ^(bool x, bool y);
```

10 The result of $x \ \& \ y$ is `true` if both x and y are `true`. Otherwise, the result is `false`.

11 The result of $x \ | \ y$ is `true` if either x or y is `true`. Otherwise, the result is `false`.

12 The result of $x \ \wedge \ y$ is `true` if x is `true` and y is `false`, or x is `false` and y is `true`. Otherwise, the result
13 is `false`. When the operands are of type `bool`, the \wedge operator computes the same result as the `!=` operator.

14 **14.11 Conditional logical operators**

15 The `&&` and `||` operators are called the conditional logical operators. They are also called the “short-
16 circuiting” logical operators.

```
17     conditional-and-expression:  
18     inclusive-or-expression  
19     conditional-and-expression && inclusive-or-expression
```

```
20     conditional-or-expression:  
21     conditional-and-expression  
22     conditional-or-expression || conditional-and-expression
```

23 The `&&` and `||` operators are conditional versions of the `&` and `|` operators:

- 24 • The operation $x \ \&\& \ y$ corresponds to the operation $x \ \& \ y$, except that y is evaluated only if x is `true`.
- 25 • The operation $x \ \|\| \ y$ corresponds to the operation $x \ | \ y$, except that y is evaluated only if x is
26 `false`.

27 An operation of the form $x \ \&\& \ y$ or $x \ \|\| \ y$ is processed by applying overload resolution (§14.2.4) as if the
28 operation was written $x \ \& \ y$ or $x \ | \ y$. Then,

- 29 • If overload resolution fails to find a single best operator, or if overload resolution selects one of the
30 predefined integer logical operators, a compile-time error occurs.
- 31 • Otherwise, if the selected operator is one of the predefined boolean logical operators (§14.10.2), the
32 operation is processed as described in §14.11.1.
- 33 • Otherwise, the selected operator is a user-defined operator, and the operation is processed as described
34 in §14.11.2.

35 It is not possible to directly overload the conditional logical operators. However, because the conditional
36 logical operators are evaluated in terms of the regular logical operators, overloads of the regular logical
37 operators are, with certain restrictions, also considered overloads of the conditional logical operators. This is
38 described further in §14.11.2.

39 **14.11.1 Boolean conditional logical operators**

40 When the operands of `&&` or `||` are of type `bool`, or when the operands are of types that do not define an
41 applicable operator `&` or operator `|`, but do define implicit conversions to `bool`, the operation is
42 processed as follows:

- 1 • The operation `x && y` is evaluated as `x ? y : false`. In other words, `x` is first evaluated and
2 converted to type `bool`. Then, if `x` is `true`, `y` is evaluated and converted to type `bool`, and this becomes
3 the result of the operation. Otherwise, the result of the operation is `false`.
- 4 • The operation `x || y` is evaluated as `x ? true : y`. In other words, `x` is first evaluated and converted
5 to type `bool`. Then, if `x` is `true`, the result of the operation is `true`. Otherwise, `y` is evaluated and
6 converted to type `bool`, and this becomes the result of the operation.

7 14.11.2 User-defined conditional logical operators

8 When the operands of `&&` or `||` are of types that declare an applicable user-defined operator `&` or
9 operator `|`, both of the following must be true, where `T` is the type in which the selected operator is
10 declared:

- 11 • The return type and the type of each parameter of the selected operator must be `T`. In other words, the
12 operator must compute the logical AND or the logical OR of two operands of type `T`, and must return a
13 result of type `T`.
- 14 • `T` must contain declarations of operator `true` and operator `false`.

15 A compile-time error occurs if either of these requirements is not satisfied. Otherwise, the `&&` or `||`
16 operation is evaluated by combining the user-defined operator `true` or operator `false` with the
17 selected user-defined operator:

- 18 • The operation `x && y` is evaluated as `T.false(x) ? x : T.&(x, y)`, where `T.false(x)` is an
19 invocation of the operator `false` declared in `T`, and `T.&(x, y)` is an invocation of the selected
20 operator `&`. In other words, `x` is first evaluated and operator `false` is invoked on the result to
21 determine if `x` is definitely false. Then, if `x` is definitely false, the result of the operation is the value
22 previously computed for `x`. Otherwise, `y` is evaluated, and the selected operator `&` is invoked on the
23 value previously computed for `x` and the value computed for `y` to produce the result of the operation.
- 24 • The operation `x || y` is evaluated as `T.true(x) ? x : T.|(x, y)`, where `T.true(x)` is an
25 invocation of the operator `true` declared in `T`, and `T.|(x, y)` is an invocation of the selected
26 operator `|`. In other words, `x` is first evaluated and operator `true` is invoked on the result to
27 determine if `x` is definitely true. Then, if `x` is definitely true, the result of the operation is the value
28 previously computed for `x`. Otherwise, `y` is evaluated, and the selected operator `|` is invoked on the
29 value previously computed for `x` and the value computed for `y` to produce the result of the operation.

30 In either of these operations, the expression given by `x` is only evaluated once, and the expression given by `y`
31 is either not evaluated or evaluated exactly once.

32 For an example of a type that implements operator `true` and operator `false`, see §18.4.2.

33 14.12 Conditional operator

34 The `?:` operator is called the conditional operator. It is at times also called the ternary operator.

35 *conditional-expression:*

36 *conditional-or-expression*

37 *conditional-or-expression ? expression : expression*

38 A conditional expression of the form `b ? x : y` first evaluates the condition `b`. Then, if `b` is `true`, `x` is
39 evaluated and becomes the result of the operation. Otherwise, `y` is evaluated and becomes the result of the
40 operation. A conditional expression never evaluates both `x` and `y`.

41 The conditional operator is right-associative, meaning that operations are grouped from right to left. For
42 example, an expression of the form `a ? b : c ? d : e` is evaluated as `a ? b : (c ? d : e)`.

43 The first operand of the `?:` operator must be an expression of a type that can be implicitly converted to
44 `bool`, or an expression of a type that implements operator `true`. If neither of these requirements is
45 satisfied, a compile-time error occurs.

1 The second and third operands of the `? :` operator control the type of the conditional expression. Let `X` and `Y`
2 be the types of the second and third operands. Then,

- 3 • If `X` and `Y` are the same type, then this is the type of the conditional expression.
- 4 • Otherwise, if an implicit conversion (§13.1) exists from `X` to `Y`, but not from `Y` to `X`, then `Y` is the type of
5 the conditional expression.
- 6 • Otherwise, if an implicit conversion (§13.1) exists from `Y` to `X`, but not from `X` to `Y`, then `X` is the type of
7 the conditional expression.
- 8 • Otherwise, no expression type can be determined, and a compile-time error occurs.

9 The run-time processing of a conditional expression of the form `b ? x : y` consists of the following steps:

- 10 • First, `b` is evaluated, and the `bool` value of `b` is determined:
 - 11 ○ If an implicit conversion from the type of `b` to `bool` exists, then this implicit conversion is
12 performed to produce a `bool` value.
 - 13 ○ Otherwise, the operator `true` defined by the type of `b` is invoked to produce a `bool` value.
- 14 • If the `bool` value produced by the step above is `true`, then `x` is evaluated and converted to the type of
15 the conditional expression, and this becomes the result of the conditional expression.
- 16 • Otherwise, `y` is evaluated and converted to the type of the conditional expression, and this becomes the
17 result of the conditional expression.

18 14.13 Assignment operators

19 The assignment operators assign a new value to a variable, a property, event, or an indexer element.

20 *assignment:*

21 *unary-expression assignment-operator expression*

22 *assignment-operator: one of*

23 `=` `+=` `-=` `*=` `/=` `%=` `&=` `|=` `^=` `<<=` `>>=`

24 The left operand of an assignment must be an expression classified as a variable, a property access, an
25 indexer access, or an event access.

26 The `=` operator is called the *simple assignment operator*. It assigns the value of the right operand to the
27 variable, property, or indexer element given by the left operand. The left operand of the simple assignment
28 operator may not be an event access (except as described in §17.7.1). The simple assignment operator is
29 described in §14.13.1.

30 The operators formed by prefixing a binary operator with an `=` character are called the *compound*
31 *assignment operators*. These operators perform the indicated operation on the two operands, and then
32 assign the resulting value to the variable, property, or indexer element given by the left operand. The
33 compound assignment operators are described in §14.13.2.

34 The `+=` and `-=` operators with an event access expression as the left operand are called the *event*
35 *assignment operators*. No other assignment operator is valid with an event access as the left operand. The
36 event assignment operators are described in §14.13.3.

37 The assignment operators are right-associative, meaning that operations are grouped from right to left. For
38 example, an expression of the form `a = b = c` is evaluated as `a = (b = c)`.

39 14.13.1 Simple assignment

40 The `=` operator is called the simple assignment operator. In a simple assignment, the right operand must be
41 an expression of a type that is implicitly convertible to the type of the left operand. The operation assigns the
42 value of the right operand to the variable, property, or indexer element given by the left operand.

1 The result of a simple assignment expression is the value assigned to the left operand. The result has the
2 same type as the left operand and is always classified as a value.

3 If the left operand is a property or indexer access, the property or indexer must have a `set` accessor. If this is
4 not the case, a compile-time error occurs.

5 The run-time processing of a simple assignment of the form `x = y` consists of the following steps:

- 6 • If `x` is classified as a variable:
 - 7 ○ `x` is evaluated to produce the variable.
 - 8 ○ `y` is evaluated and, if required, converted to the type of `x` through an implicit conversion (§13.1).
 - 9 ○ If the variable given by `x` is an array element of a *reference-type*, a run-time check is performed to
10 ensure that the value computed for `y` is compatible with the array instance of which `x` is an element.
11 The check succeeds if `y` is `null`, or if an implicit reference conversion (§13.1.4) exists from the
12 actual type of the instance referenced by `y` to the actual element type of the array instance containing
13 `x`. Otherwise, a `System.ArrayTypeMismatchException` is thrown.
 - 14 ○ The value resulting from the evaluation and conversion of `y` is stored into the location given by the
15 evaluation of `x`.
- 16 • If `x` is classified as a property or indexer access:
 - 17 ○ The instance expression (if `x` is not `static`) and the argument list (if `x` is an indexer access)
18 associated with `x` are evaluated, and the results are used in the subsequent `set` accessor invocation.
 - 19 ○ `y` is evaluated and, if required, converted to the type of `x` through an implicit conversion (§13.1).
 - 20 ○ The `set` accessor of `x` is invoked with the value computed for `y` as its `value` argument.

21 [Note: The array covariance rules (§19.5) permit a value of an array type `A[]` to be a reference to an
22 instance of an array type `B[]`, provided an implicit reference conversion exists from `B` to `A`. Because of
23 these rules, assignment to an array element of a *reference-type* requires a run-time check to ensure that the
24 value being assigned is compatible with the array instance. In the example

```
25     string[] sa = new string[10];
26     object[] oa = sa;
27     oa[0] = null;           // Ok
28     oa[1] = "Hello";      // Ok
29     oa[2] = new ArrayList(); // ArrayTypeMismatchException
```

30 the last assignment causes a `System.ArrayTypeMismatchException` to be thrown because an instance
31 of `ArrayList` cannot be stored in an element of a `string[]`. *end note*

32 When a property or indexer declared in a *struct-type* is the target of an assignment, the instance expression
33 associated with the property or indexer access must be classified as a variable. If the instance expression is
34 classified as a value, a compile-time error occurs. [Note: Because of §14.5.4, the same rule also applies to
35 fields. *end note*]

36 [Example: Given the declarations:

```
37     struct Point
38     {
39         int x, y;
40         public Point(int x, int y) {
41             this.x = x;
42             this.y = y;
43         }
44         public int X {
45             get { return x; }
46             set { x = value; }
47         }

```

```

1      public int Y {
2          get { return y; }
3          set { y = value; }
4      }
5  }
6  struct Rectangle
7  {
8      Point a, b;
9      public Rectangle(Point a, Point b) {
10         this.a = a;
11         this.b = b;
12     }
13     public Point A {
14         get { return a; }
15         set { a = value; }
16     }
17     public Point B {
18         get { return b; }
19         set { b = value; }
20     }
21 }

```

22 in the example

```

23     Point p = new Point();
24     p.X = 100;
25     p.Y = 100;
26     Rectangle r = new Rectangle();
27     r.A = new Point(10, 10);
28     r.B = p;

```

29 the assignments to `p.X`, `p.Y`, `r.A`, and `r.B` are permitted because `p` and `r` are variables. However, in the
30 example

```

31     Rectangle r = new Rectangle();
32     r.A.X = 10;
33     r.A.Y = 10;
34     r.B.X = 100;
35     r.B.Y = 100;

```

36 the assignments are all invalid, since `r.A` and `r.B` are not variables. *end example*]

37 14.13.2 Compound assignment

38 An operation of the form `x op= y` is processed by applying binary operator overload resolution (§14.2.4) as
39 if the operation was written `x op y`. Then,

- 40 • If the return type of the selected operator is *implicitly* convertible to the type of `x`, the operation is
41 evaluated as `x = x op y`, except that `x` is evaluated only once.
- 42 • Otherwise, if the selected operator is a predefined operator, if the return type of the selected operator is
43 *explicitly* convertible to the type of `x`, and if `y` is *implicitly* convertible to the type of `x`, then the
44 operation is evaluated as `x = (T)(x op y)`, where `T` is the type of `x`, except that `x` is evaluated only
45 once.
- 46 • Otherwise, the compound assignment is invalid, and a compile-time error occurs.

47 The term “evaluated only once” means that in the evaluation of `x op y`, the results of any constituent
48 expressions of `x` are temporarily saved and then reused when performing the assignment to `x`. [*Example*: For
49 example, in the assignment `A() [B()] += C()`, where `A` is a method returning `int[]`, and `B` and `C` are
50 methods returning `int`, the methods are invoked only once, in the order `A, B, C`. *end example*]

51 When the left operand of a compound assignment is a property access or indexer access, the property or
52 indexer must have both a `get` accessor and a `set` accessor. If this is not the case, a compile-time error
53 occurs.

1 The second rule above permits `x op= y` to be evaluated as `x = (T)(x op y)` in certain contexts. The rule
 2 exists such that the predefined operators can be used as compound operators when the left operand is of type
 3 `sbyte`, `byte`, `short`, `ushort`, or `char`. Even when both arguments are of one of those types, the
 4 predefined operators produce a result of type `int`, as described in §14.2.6.2. Thus, without a cast it would
 5 not be possible to assign the result to the left operand.

6 The intuitive effect of the rule for predefined operators is simply that `x op= y` is permitted if both of
 7 `x op y` and `x = y` are permitted. [Example: In the example

```

8     byte b = 0;
9     char ch = '\0';
10    int i = 0;
11
12    b += 1;           // Ok
13    b += 1000;       // Error, b = 1000 not permitted
14    b += i;          // Error, b = i not permitted
15    b += (byte)i;    // Ok
16
17    ch += 1;         // Error, ch = 1 not permitted
18    ch += (char)1;   // Ok

```

19 the intuitive reason for each error is that a corresponding simple assignment would also have been an error.
 20 *end example]*

19 14.13.3 Event assignment

20 If the left operand of a `+=` or `-=` operator is classified as an event access, then the expression is evaluated as
 21 follows:

- 22 • The instance expression, if any, of the event access is evaluated.
- 23 • The right operand of the `+=` or `-=` operator is evaluated, and, if required, converted to the type of the left
 24 operand through an implicit conversion (§13.1).
- 25 • An event accessor of the event is invoked, with argument list consisting of the right operand, after
 26 evaluation and, if necessary, conversion. If the operator was `+=`, the `add` accessor is invoked; if the
 27 operator was `-=`, the `remove` accessor is invoked.

28 An event assignment expression does not yield a value. Thus, an event assignment expression is valid only
 29 in the context of a *statement-expression* (§15.6).

30 14.14 Expression

31 An *expression* is either a *conditional-expression* or an *assignment*.

```

32     expression:
33         conditional-expression
34         assignment

```

35 14.15 Constant expressions

36 A *constant-expression* is an expression that can be fully evaluated at compile-time.

```

37     constant-expression:
38         expression

```

39 The type of a constant expression can be one of the following: `sbyte`, `byte`, `short`, `ushort`, `int`, `uint`,
 40 `long`, `ulong`, `char`, `float`, `double`, `decimal`, `bool`, `string`, any enumeration type, or the null type.
 41 The following constructs are permitted in constant expressions:

- 42 • Literals (including the `null` literal).
- 43 • References to `const` members of class and struct types.
- 44 • References to members of enumeration types.

- 1 • Parenthesized sub-expressions, which are themselves constant expressions.
- 2 • Cast expressions, provided the target type is one of the types listed above.
- 3 • The predefined `+`, `-`, `!`, and `~` unary operators.
- 4 • The predefined `+`, `-`, `*`, `/`, `%`, `<<`, `>>`, `&`, `|`, `^`, `&&`, `||`, `==`, `!=`, `<`, `>`, `<=`, and `>=` binary operators, provided
- 5 each operand is of a type listed above.
- 6 • The `?:` conditional operator.

7 Whenever an expression is of one of the types listed above and contains only the constructs listed above, the
8 expression is evaluated at compile-time. This is true even if the expression is a sub-expression of a larger
9 expression that contains non-constant constructs.

10 The compile-time evaluation of constant expressions uses the same rules as run-time evaluation of non-
11 constant expressions, except that where run-time evaluation would have thrown an exception, compile-time
12 evaluation causes a compile-time error to occur.

13 Unless a constant expression is explicitly placed in an **unchecked** context, overflows that occur in integral-
14 type arithmetic operations and conversions during the compile-time evaluation of the expression always
15 cause compile-time errors (§14.5.12).

16 Constant expressions occur in the contexts listed below. In these contexts, a compile-time error occurs if an
17 expression cannot be fully evaluated at compile-time.

- 18 • Constant declarations (§17.3).
- 19 • Enumeration member declarations (§21.30).
- 20 • `case` labels of a `switch` statement (§15.7.2).
- 21 • `goto case` statements (§15.9.3).
- 22 • Dimension lengths in an array creation expression (§14.5.10.2) that includes an initializer.
- 23 • Attributes (§24).

24 An implicit constant expression conversion (§13.1.6) permits a constant expression of type `int` to be
25 converted to `sbyte`, `byte`, `short`, `ushort`, `uint`, or `ulong`, provided the value of the constant expression
26 is within the range of the destination type.

27 **14.16 Boolean expressions**

28 A *boolean-expression* is an expression that yields a result of type `bool`.

29 *boolean-expression:*
30 *expression*

31 The controlling conditional expression of an *if-statement* (§15.7.1), *while-statement* (§15.8.1), *do-statement*
32 (§15.8.2), or *for-statement* (§15.8.3) is a *boolean-expression*. The controlling conditional expression of the
33 `?:` operator (§14.12) follows the same rules as a *boolean-expression*, but for reasons of operator precedence
34 is classified as a *conditional-or-expression*.

35 A *boolean-expression* is required to be of a type that can be implicitly converted to `bool` or of a type that
36 implements `operator true`. [Note: As required by §17.9.1, any type that implements `operator true`
37 must also implement `operator false`. end note] If neither requirement is satisfied, a compile-time error
38 occurs.

39 When a boolean expression is of a type that cannot be implicitly converted to `bool` but does implement
40 `operator true`, then following evaluation of the expression, the `operator true` implementation
41 provided by that type is invoked to produce a `bool` value.

42 [Note: The `DBBool` struct type in §18.4.2 provides an example of a type that implements `operator true`
43 and `operator false`. end note]

15. Statements

1

2 C# provides a variety of statements. [*Note: Most of these statements will be familiar to developers who have*
3 *programmed in C and C++. end note*]

4 *statement:*
5 *labeled-statement*
6 *declaration-statement*
7 *embedded-statement*

8 *embedded-statement:*
9 *block*
10 *empty-statement*
11 *expression-statement*
12 *selection-statement*
13 *iteration-statement*
14 *jump-statement*
15 *try-statement*
16 *checked-statement*
17 *unchecked-statement*
18 *lock-statement*
19 *using-statement*

20 The *embedded-statement* nonterminal is used for statements that appear within other statements. The use of
21 *embedded-statement* rather than *statement* excludes the use of declaration statements and labeled statements
22 in these contexts. [*Example: The code*

```
23     void F(bool b) {
24         if (b)
25             int i = 44;
26     }
```

27 results in a compile-time error because an `if` statement requires an *embedded-statement* rather than a
28 *statement* for its if branch. If this code were permitted, then the variable `i` would be declared, but it could
29 never be used. (Note, however, that by placing `i`'s declaration in a block, the example is valid.) *end*
30 *example*]

31 15.1 End points and reachability

32 Every statement has an *end point*. In intuitive terms, the end point of a statement is the location that
33 immediately follows the statement. The execution rules for composite statements (statements that contain
34 embedded statements) specify the action that is taken when control reaches the end point of an embedded
35 statement. For example, when control reaches the end point of a statement in a block, control is transferred
36 to the next statement in the block.

37 If a statement can possibly be reached by execution, the statement is said to be *reachable*. Conversely, if
38 there is no possibility that a statement will be executed, the statement is said to be *unreachable*.

39 [*Example: In the example*

```
40     void F() {
41         Console.WriteLine("reachable");
42         goto Label;
43         Console.WriteLine("unreachable");
44         Label:
45         Console.WriteLine("reachable");
46     }
```

C# LANGUAGE SPECIFICATION

1 the second invocation of `Console.WriteLine` is unreachable because there is no possibility that the
2 statement will be executed. *end example*]

3 A warning is reported if the compiler determines that a statement is unreachable. It is specifically not an
4 error for a statement to be unreachable.

5 [Note: To determine whether a particular statement or end point is reachable, the compiler performs flow
6 analysis according to the reachability rules defined for each statement. The flow analysis takes into account
7 the values of constant expressions (§14.15) that control the behavior of statements, but the possible values of
8 non-constant expressions are not considered. In other words, for purposes of control flow analysis, a non-
9 constant expression of a given type is considered to have any possible value of that type.

10 In the example

```
11     void F() {  
12         const int i = 1;  
13         if (i == 2) Console.WriteLine("unreachable");  
14     }
```

15 the boolean expression of the `if` statement is a constant expression because both operands of the
16 `==` operator are constants. As the constant expression is evaluated at compile-time, producing the value
17 `false`, the `Console.WriteLine` invocation is considered unreachable. However, if `i` is changed to be a
18 local variable

```
19     void F() {  
20         int i = 1;  
21         if (i == 2) Console.WriteLine("reachable");  
22     }
```

23 the `Console.WriteLine` invocation is considered reachable, even though, in reality, it will never be
24 executed. *end note*]

25 The *block* of a function member is always considered reachable. By successively evaluating the reachability
26 rules of each statement in a block, the reachability of any given statement can be determined.

27 [Example: In the example

```
28     void F(int x) {  
29         Console.WriteLine("start");  
30         if (x < 0) Console.WriteLine("negative");  
31     }
```

32 the reachability of the second `Console.WriteLine` is determined as follows:

- 33 • The first `Console.WriteLine` expression statement is reachable because the block of the `F` method is
34 reachable (§15.2).
- 35 • The end point of the first `Console.WriteLine` expression statement is reachable^{15.2} because that
36 statement is reachable (§15.6 and §15.2).
- 37 • The `if` statement is reachable because the end point of the first `Console.WriteLine` expression
38 statement is reachable (§15.6 and §15.2).
- 39 • The second `Console.WriteLine` expression statement is reachable because the boolean expression of
40 the `if` statement does not have the constant value `false`.

41 *end example*]

42 There are two situations in which it is a compile-time error for the end point of a statement to be reachable:

- 43 • Because the `switch` statement does not permit a switch section to “fall through” to the next switch
44 section, it is a compile-time error for the end point of the statement list of a switch section to be
45 reachable. If this error occurs, it is typically an indication that a `break` statement is missing.
- 46 • It is a compile-time error for the end point of the block of a function member that computes a value to be
47 reachable. If this error occurs, it typically is an indication that a `return` statement is missing.

1 15.2 Blocks

2 A *block* permits multiple statements to be written in contexts where a single statement is allowed.

```
3     block:
4         { statement-listopt }
```

5 A *block* consists of an optional *statement-list* (§15.2.1), enclosed in braces. If the statement list is omitted,
6 the block is said to be empty.

7 A block may contain declaration statements (§15.5). The scope of a local variable or constant declared in a
8 block is the block.

9 Within a block, the meaning of a name used in an expression context must always be the same (§14.5.2.1).

10 A block is executed as follows:

- 11 • If the block is empty, control is transferred to the end point of the block.
- 12 • If the block is not empty, control is transferred to the statement list. When and if control reaches the end
13 point of the statement list, control is transferred to the end point of the block.

14 The statement list of a block is reachable if the block itself is reachable.

15 The end point of a block is reachable if the block is empty or if the end point of the statement list is
16 reachable.

17 15.2.1 Statement lists

18 A *statement list* consists of one or more statements written in sequence. Statement lists occur in *blocks*
19 (§15.2) and in *switch-blocks* (§15.7.2).

```
20     statement-list:
21         statement
22         statement-list statement
```

23 A statement list is executed by transferring control to the first statement. When and if control reaches the end
24 point of a statement, control is transferred to the next statement. When and if control reaches the end point of
25 the last statement, control is transferred to the end point of the statement list.

26 A statement in a statement list is reachable if at least one of the following is true:

- 27 • The statement is the first statement and the statement list itself is reachable.
- 28 • The end point of the preceding statement is reachable.
- 29 • The statement is a labeled statement and the label is referenced by a reachable `goto` statement.

30 The end point of a statement list is reachable if the end point of the last statement in the list is reachable.

31 15.3 The empty statement

32 An *empty-statement* does nothing.

```
33     empty-statement:
34         ;
```

35 An empty statement is used when there are no operations to perform in a context where a statement is
36 required.

37 Execution of an empty statement simply transfers control to the end point of the statement. Thus, the end
38 point of an empty statement is reachable if the empty statement is reachable.

39 [Example: An empty statement can be used when writing a `while` statement with a null body:

```
40     bool ProcessMessage() {...}
```

C# LANGUAGE SPECIFICATION

```
1     void ProcessMessages() {
2         while (ProcessMessage())
3             ;
4     }
```

5 Also, an empty statement can be used to declare a label just before the closing “}” of a block:

```
6     void F() {
7         ...
8         if (done) goto exit;
9         ...
10        exit: ;
11    }
```

12 *end example]*

13 15.4 Labeled statements

14 A *labeled-statement* permits a statement to be prefixed by a label. Labeled statements are permitted in
15 blocks, but are not permitted as embedded statements.

```
16     labeled-statement:  
17     identifier : statement
```

18 A labeled statement declares a label with the name given by the *identifier*. The scope of a label is the whole
19 block in which the label is declared, including any nested blocks. It is a compile-time error for two labels
20 with the same name to have overlapping scopes.

21 A label can be referenced from `goto` statements (§15.9.3) within the scope of the label. [*Note:* This means
22 that `goto` statements can transfer control within blocks and out of blocks, but never into blocks. *end note]*

23 Labels have their own declaration space and do not interfere with other identifiers. [*Example:* The example

```
24     int F(int x) {
25         if (x >= 0) goto x;
26         x = -x;
27         x: return x;
28     }
```

29 is valid and uses the name `x` as both a parameter and a label. *end example]*

30 Execution of a labeled statement corresponds exactly to execution of the statement following the label.

31 In addition to the reachability provided by normal flow of control, a labeled statement is reachable if the
32 label is referenced by a reachable `goto` statement. (Exception: If a `goto` statement is inside a `try` that
33 includes a `finally` block, and the labeled statement is outside the `try`, and the end point of the `finally`
34 block is unreachable, then the labeled statement is not reachable from that `goto` statement.)

35 15.5 Declaration statements

36 A *declaration-statement* declares a local variable or constant. Declaration statements are permitted in blocks,
37 but are not permitted as embedded statements.

```
38     declaration-statement:  
39     local-variable-declaration ;  
40     local-constant-declaration ;
```

41 15.5.1 Local variable declarations

42 A *local-variable-declaration* declares one or more local variables.

```
43     local-variable-declaration:  
44     type local-variable-declarators
```

```

1      local-variable-declarators:
2          local-variable-declarator
3          local-variable-declarators , local-variable-declarator
4
5      local-variable-declarator:
6          identifier
7          identifier = local-variable-initializer
8
9      local-variable-initializer:
10         expression
11         array-initializer

```

10 The *type* of a *local-variable-declaration* specifies the type of the variables introduced by the declaration.
 11 The type is followed by a list of *local-variable-declarators*, each of which introduces a new variable. A
 12 *local-variable-declarator* consists of an *identifier* that names the variable, optionally followed by an
 13 “=” token and a *local-variable-initializer* that gives the initial value of the variable.

14 The value of a local variable is obtained in an expression using a *simple-name* (§14.5.2), and the value of a
 15 local variable is modified using an *assignment* (§14.13). A local variable must be definitely assigned (§12.3)
 16 at each location where its value is obtained.

17 The scope of a local variable declared in a *local-variable-declaration* is the block in which the declaration
 18 occurs. It is an error to refer to a local variable in a textual position that precedes the *local-variable-*
 19 *declarator* of the local variable. Within the scope of a local variable, it is a compile-time error to declare
 20 another local variable or constant with the same name.

21 A local variable declaration that declares multiple variables is equivalent to multiple declarations of single
 22 variables with the same type. Furthermore, a variable initializer in a local variable declaration corresponds
 23 exactly to an assignment statement that is inserted immediately after the declaration.

24 [*Example:* The example

```

25     void F() {
26         int x = 1, y, z = x * 2;
27     }

```

28 corresponds exactly to

```

29     void F() {
30         int x; x = 1;
31         int y;
32         int z; z = x * 2;
33     }

```

34 *end example*]

35 15.5.2 Local constant declarations

36 A *local-constant-declaration* declares one or more local constants.

```

37     local-constant-declaration:
38         const type constant-declarators
39
40     constant-declarators:
41         constant-declarator
42         constant-declarators , constant-declarator
43
44     constant-declarator:
45         identifier = constant-expression

```

44 The *type* of a *local-constant-declaration* specifies the type of the constants introduced by the declaration.
 45 The type is followed by a list of *constant-declarators*, each of which introduces a new constant. A *constant-*
 46 *declarator* consists of an *identifier* that names the constant, followed by an “=” token, followed by a
 47 *constant-expression* (§14.15) that gives the value of the constant.

1 The *type* and *constant-expression* of a local constant declaration must follow the same rules as those of a
2 constant member declaration (§17.3).

3 The value of a local constant is obtained in an expression using a *simple-name* (§14.5.2).

4 The scope of a local constant is the block in which the declaration occurs. It is an error to refer to a local
5 constant in a textual position that precedes its *constant-declarator*. Within the scope of a local constant, it is
6 a compile-time error to declare another local variable or constant with the same name.

7 A local constant declaration that declares multiple constants is equivalent to multiple declarations of single
8 constants with the same type.

9 **15.6 Expression statements**

10 An *expression-statement* evaluates a given expression. The value computed by the expression, if any, is
11 discarded.

12 *expression-statement:*
13 *statement-expression* ;

14 *statement-expression:*
15 *invocation-expression*
16 *object-creation-expression*
17 *assignment*
18 *post-increment-expression*
19 *post-decrement-expression*
20 *pre-increment-expression*
21 *pre-decrement-expression*

22 Not all expressions are permitted as statements. [*Note:* In particular, expressions such as $x + y$ and
23 $x == 1$, that merely compute a value (which will be discarded), are not permitted as statements. *end note*]

24 Execution of an expression statement evaluates the contained expression and then transfers control to the
25 end point of the expression statement. The end point of an *expression-statement* is reachable if that
26 *expression-statement* is reachable.

27 **15.7 Selection statements**

28 Selection statements select one of a number of possible statements for execution based on the value of some
29 expression.

30 *selection-statement:*
31 *if-statement*
32 *switch-statement*

33 **15.7.1 The if statement**

34 The *if* statement selects a statement for execution based on the value of a boolean expression.

35 *if-statement:*
36 *if* (*boolean-expression*) *embedded-statement*
37 *if* (*boolean-expression*) *embedded-statement* *else* *embedded-statement*

38 *boolean-expression:*
39 *expression*

40 An *else* part is associated with the lexically nearest preceding *if* that is allowed by the syntax. [*Example:*
41 Thus, an *if* statement of the form

42 `if (x) if (y) F(); else G();`

43 is equivalent to

```

1     if (x) {
2         if (y) {
3             F();
4         }
5         else {
6             G();
7         }
8     }

```

9 *end example]*

10 An `if` statement is executed as follows:

- 11 • The *boolean-expression* (§14.16) is evaluated.
- 12 • If the boolean expression yields `true`, control is transferred to the first embedded statement. When and
13 if control reaches the end point of that statement, control is transferred to the end point of the `if`
14 statement.
- 15 • If the boolean expression yields `false` and if an `else` part is present, control is transferred to the
16 second embedded statement. When and if control reaches the end point of that statement, control is
17 transferred to the end point of the `if` statement.
- 18 • If the boolean expression yields `false` and if an `else` part is not present, control is transferred to the
19 end point of the `if` statement.

20 The first embedded statement of an `if` statement is reachable if the `if` statement is reachable and the
21 boolean expression does not have the constant value `false`.

22 The second embedded statement of an `if` statement, if present, is reachable if the `if` statement is reachable
23 and the boolean expression does not have the constant value `true`.

24 The end point of an `if` statement is reachable if the end point of at least one of its embedded statements is
25 reachable. In addition, the end point of an `if` statement with no `else` part is reachable if the `if` statement is
26 reachable and the boolean expression does not have the constant value `true`.

27 15.7.2 The `switch` statement

28 The `switch` statement selects for execution a statement list having an associated switch label that corresponds
29 to the value of the switch expression.

```

30     switch-statement:
31     switch ( expression ) switch-block

```

```

32     switch-block:
33     { switch-sectionsopt }

```

```

34     switch-sections:
35     switch-section
36     switch-sections switch-section

```

```

37     switch-section:
38     switch-labels statement-list

```

```

39     switch-labels:
40     switch-label
41     switch-labels switch-label

```

```

42     switch-label:
43     case constant-expression :
44     default :

```

45 A *switch-statement* consists of the keyword `switch`, followed by a parenthesized expression (called the
46 switch expression), followed by a *switch-block*. The *switch-block* consists of zero or more *switch-sections*,

1 enclosed in braces. Each *switch-section* consists of one or more *switch-labels* followed by a *statement-list*
2 (§15.2.1).

3 The *governing type* of a `switch` statement is established by the switch expression. If the type of the switch
4 expression is `sbyte`, `byte`, `short`, `ushort`, `int`, `uint`, `long`, `ulong`, `char`, `string`, or an *enum-type*,
5 then that is the governing type of the `switch` statement. Otherwise, exactly one user-defined implicit
6 conversion (§13.4) must exist from the type of the switch expression to one of the following possible
7 governing types: `sbyte`, `byte`, `short`, `ushort`, `int`, `uint`, `long`, `ulong`, `char`, `string`. If no such
8 implicit conversion exists, or if more than one such implicit conversion exists, a compile-time error occurs.

9 The constant expression of each `case` label must denote a value of a type that is implicitly convertible
10 (§13.1) to the governing type of the `switch` statement. A compile-time error occurs if two or more `case`
11 labels in the same `switch` statement specify the same constant value.

12 There can be at most one `default` label in a switch statement.

13 A `switch` statement is executed as follows:

- 14 • The switch expression is evaluated and converted to the governing type.
- 15 • If one of the constants specified in a `case` label in the same `switch` statement is equal to the value of
16 the switch expression, control is transferred to the statement list following the matched `case` label.
- 17 • If none of the constants specified in `case` labels in the same `switch` statement is equal to the value of
18 the switch expression, and if a `default` label is present, control is transferred to the statement list
19 following the `default` label.
- 20 • If none of the constants specified in `case` labels in the same `switch` statement is equal to the value of
21 the switch expression, and if no `default` label is present, control is transferred to the end point of the
22 `switch` statement.

23 If the end point of the statement list of a switch section is reachable, a compile-time error occurs. This is
24 known as the “no fall through” rule. [*Example:* The example

```
25     switch (i) {
26     case 0:
27         CaseZero();
28         break;
29     case 1:
30         CaseOne();
31         break;
32     default:
33         CaseOthers();
34         break;
35     }
```

36 is valid because no switch section has a reachable end point. Unlike C and C++, execution of a switch
37 section is not permitted to “fall through” to the next switch section, and the example

```
38     switch (i) {
39     case 0:
40         CaseZero();
41     case 1:
42         CaseZeroOrOne();
43     default:
44         CaseAny();
45     }
```

46 results in a compile-time error. When execution of a switch section is to be followed by execution of another
47 switch section, an explicit `goto case` or `goto default` statement must be used:

```

1      switch (i) {
2      case 0:
3          CaseZero();
4          goto case 1;
5      case 1:
6          CaseZeroOrOne();
7          goto default;
8      default:
9          CaseAny();
10         break;
11     }

```

12 *end example]*

13 Multiple labels are permitted in a *switch-section*. [*Example:* The example

```

14     switch (i) {
15     case 0:
16         CaseZero();
17         break;
18     case 1:
19         CaseOne();
20         break;
21     case 2:
22     default:
23         CaseTwo();
24         break;
25     }

```

26 is valid. The example does not violate the “no fall through” rule because the labels `case 2:` and `default:`
27 are part of the same *switch-section*. *end example]*

28 [*Note:* The “no fall through” rule prevents a common class of bugs that occur in C and C++ when `break`
29 statements are accidentally omitted. In addition, because of this rule, the `switch` sections of a `switch`
30 statement can be arbitrarily rearranged without affecting the behavior of the statement. For example, the
31 sections of the `switch` statement above can be reversed without affecting the behavior of the statement:

```

32     switch (i) {
33     default:
34         CaseAny();
35         break;
36     case 1:
37         CaseZeroOrOne();
38         goto default;
39     case 0:
40         CaseZero();
41         goto case 1;
42     }

```

43 *end note]*

44 [*Note:* The statement list of a `switch` section typically ends in a `break`, `goto case`, or `goto default`
45 statement, but any construct that renders the end point of the statement list unreachable is permitted. For
46 example, a `while` statement controlled by the boolean expression `true` is known to never reach its end
47 point. Likewise, a `throw` or `return` statement always transfers control elsewhere and never reaches its end
48 point. Thus, the following example is valid:

```

49     switch (i) {
50     case 0:
51         while (true) F();
52     case 1:
53         throw new ArgumentException();
54     case 2:
55         return;
56     }

```

57 *end note]*

58 [*Example:* The governing type of a `switch` statement may be the type `string`. For example:

```
1     void DoCommand(string command) {
2         switch (command.ToLower()) {
3             case "run":
4                 DoRun();
5                 break;
6             case "save":
7                 DoSave();
8                 break;
9             case "quit":
10                DoQuit();
11                break;
12            default:
13                InvalidCommand(command);
14                break;
15        }
16    }
```

17 *end example]*

18 [Note: Like the string equality operators (§14.9.7), the `switch` statement is case sensitive and will execute a
19 given switch section only if the switch expression string exactly matches a `case` label constant. *end note]*

20 When the governing type of a `switch` statement is `string`, the value `null` is permitted as a case label
21 constant.

22 The *statement-lists* of a *switch-block* may contain declaration statements (§15.5). The scope of a local
23 variable or constant declared in a switch block is the switch block.

24 Within a switch block, the meaning of a name used in an expression context must always be the same
25 (§14.5.2.1).

26 The statement list of a given switch section is reachable if the `switch` statement is reachable and at least
27 one of the following is true:

- 28 • The switch expression is a non-constant value.
- 29 • The switch expression is a constant value that matches a `case` label in the switch section.
- 30 • The switch expression is a constant value that doesn't match any `case` label, and the switch section
31 contains the `default` label.
- 32 • A switch label of the switch section is referenced by a reachable `goto case` or `goto default`
33 statement.

34 The end point of a `switch` statement is reachable if at least one of the following is true:

- 35 • The `switch` statement contains a reachable `break` statement that exits the `switch` statement.
- 36 • The `switch` statement is reachable, the switch expression is a non-constant value, and no `default`
37 label is present.
- 38 • The `switch` statement is reachable, the switch expression is a constant value that doesn't match any
39 case label, and no `default` label is present.

40 15.8 Iteration statements

41 Iteration statements repeatedly execute an embedded statement.

42 *iteration-statement:*

43 *while-statement*

44 *do-statement*

45 *for-statement*

46 *foreach-statement*

47 15.8.1 The `while` statement

48 The `while` statement conditionally executes an embedded statement zero or more times.

1 *while-statement:*
 2 while (*boolean-expression*) *embedded-statement*

3 A `while` statement is executed as follows:

- 4 • The *boolean-expression* (§14.16) is evaluated.
- 5 • If the boolean expression yields `true`, control is transferred to the embedded statement. When and if
 6 control reaches the end point of the embedded statement (possibly from execution of a `continue`
 7 statement), control is transferred to the beginning of the `while` statement.
- 8 • If the boolean expression yields `false`, control is transferred to the end point of the `while` statement.

9 Within the embedded statement of a `while` statement, a `break` statement (§15.9.1) may be used to transfer
 10 control to the end point of the `while` statement (thus ending iteration of the embedded statement), and a
 11 `continue` statement (§15.9.2) may be used to transfer control to the end point of the embedded statement
 12 (thus performing another iteration of the `while` statement).

13 The embedded statement of a `while` statement is reachable if the `while` statement is reachable and the
 14 boolean expression does not have the constant value `false`.

15 The end point of a `while` statement is reachable if at least one of the following is true:

- 16 • The `while` statement contains a reachable `break` statement that exits the `while` statement.
- 17 • The `while` statement is reachable and the boolean expression does not have the constant value `true`.

18 **15.8.2 The do statement**

19 The `do` statement conditionally executes an embedded statement one or more times.

20 *do-statement:*
 21 do *embedded-statement* while (*boolean-expression*) ;

22 A `do` statement is executed as follows:

- 23 • Control is transferred to the embedded statement.
- 24 • When and if control reaches the end point of the embedded statement (possibly from execution of a
 25 `continue` statement), the *boolean-expression* (§14.16) is evaluated. If the boolean expression yields
 26 `true`, control is transferred to the beginning of the `do` statement. Otherwise, control is transferred to the
 27 end point of the `do` statement.

28 Within the embedded statement of a `do` statement, a `break` statement (§15.9.1) may be used to transfer
 29 control to the end point of the `do` statement (thus ending iteration of the embedded statement), and a
 30 `continue` statement (§15.9.2) may be used to transfer control to the end point of the embedded statement
 31 (thus performing another iteration of the `do` statement).

32 The embedded statement of a `do` statement is reachable if the `do` statement is reachable.

33 The end point of a `do` statement is reachable if at least one of the following is true:

- 34 • The `do` statement contains a reachable `break` statement that exits the `do` statement.
- 35 • The end point of the embedded statement is reachable and the boolean expression does not have the
 36 constant value `true`.

37 **15.8.3 The for statement**

38 The `for` statement evaluates a sequence of initialization expressions and then, while a condition is true,
 39 repeatedly executes an embedded statement and evaluates a sequence of iteration expressions.

40 *for-statement:*
 41 for (*for-initializer_{opt}* ; *for-condition_{opt}* ; *for-iterator_{opt}*) *embedded-statement*

1 *for-initializer*:
 2 *local-variable-declaration*
 3 *statement-expression-list*

4 *for-condition*:
 5 *boolean-expression*

6 *for-iterator*:
 7 *statement-expression-list*

8 *statement-expression-list*:
 9 *statement-expression*
 10 *statement-expression-list* , *statement-expression*

11 The *for-initializer*, if present, consists of either a *local-variable-declaration* (§15.5.1) or a list of *statement-expressions* (§15.6) separated by commas. The scope of a local variable declared by a *for-initializer* starts at the *local-variable-declarator* for the variable and extends to the end of the embedded statement. The scope includes the *for-condition* and the *for-iterator*.

15 The *for-condition*, if present, must be a *boolean-expression* (§14.16).

16 The *for-iterator*, if present, consists of a list of *statement-expressions* (§15.6) separated by commas.

17 A **for** statement is executed as follows:

- 18 • If a *for-initializer* is present, the variable initializers or statement expressions are executed in the order they are written. This step is only performed once.
- 20 • If a *for-condition* is present, it is evaluated.
- 21 • If the *for-condition* is not present or if the evaluation yields **true**, control is transferred to the embedded statement. When and if control reaches the end point of the embedded statement (possibly from execution of a **continue** statement), the expressions of the *for-iterator*, if any, are evaluated in sequence, and then another iteration is performed, starting with evaluation of the *for-condition* in the step above.
- 26 • If the *for-condition* is present and the evaluation yields **false**, control is transferred to the end point of the **for** statement.

28 Within the embedded statement of a **for** statement, a **break** statement (§15.9.1) may be used to transfer control to the end point of the **for** statement (thus ending iteration of the embedded statement), and a **continue** statement (§15.9.2) may be used to transfer control to the end point of the embedded statement (thus executing another iteration of the **for** statement).

32 The embedded statement of a **for** statement is reachable if one of the following is true:

- 33 • The **for** statement is reachable and no *for-condition* is present.
- 34 • The **for** statement is reachable and a *for-condition* is present and does not have the constant value **false**.

36 The end point of a **for** statement is reachable if at least one of the following is true:

- 37 • The **for** statement contains a reachable **break** statement that exits the **for** statement.
- 38 • The **for** statement is reachable and a *for-condition* is present and does not have the constant value **true**.

40 **15.8.4 The foreach statement**

41 The **foreach** statement enumerates the elements of a collection, executing an embedded statement for each element of the collection.

43 *foreach-statement*:
 44 **foreach** (*type identifier in expression*) *embedded-statement*

1 The *type* and *identifier* of a `foreach` statement declare the *iteration variable* of the statement. The iteration
 2 variable corresponds to a read-only local variable with a scope that extends over the embedded statement.
 3 During execution of a `foreach` statement, the iteration variable represents the collection element for which
 4 an iteration is currently being performed. A compile-time error occurs if the embedded statement attempts to
 5 modify the iteration variable (via assignment or the `++` and `--` operators) or pass the iteration variable as a
 6 `ref` or `out` parameter.

7 The type of the *expression* of a `foreach` statement must be a collection type (as defined below), and an
 8 explicit conversion (§13.2) must exist from the element type of the collection to the type of the iteration
 9 variable. If *expression* has the value `null`, a `System.NullReferenceException` is thrown.

10 A type `C` is said to be a *collection type* if it implements the `System.IEnumerable` interface or implements
 11 the *collection pattern* by meeting all of the following criteria:

- 12 • `C` contains a `public` instance method with the signature `GetEnumerator()`, that returns a *struct-type*,
 13 *class-type*, or *interface-type*, which is called `E` in the following text.
- 14 • `E` contains a `public` instance method with the signature `MoveNext()` and the return type `bool`.
- 15 • `E` contains a `public` instance property named `Current` that permits reading the current value. The type
 16 of this property is said to be the *element type* of the collection type.

17 A type that implements `IEnumerable` is also a collection type, even if it doesn't satisfy the conditions
 18 above. (This is possible if it implements `IEnumerable` via private interface implementation.)

19 The `System.Array` type (§19.1.1) is a collection type, and since all array types derive from
 20 `System.Array`, any array type expression is permitted in a `foreach` statement. The order in which
 21 `foreach` traverses the elements of an array is as follows: For single-dimensional arrays elements are
 22 traversed in increasing index order, starting with index `0` and ending with index `Length - 1`. For multi-
 23 dimensional arrays, elements are traversed such that the indices of the rightmost dimension are increased
 24 first, then the next left dimension, and so on to the left.

25 A `foreach` statement of the form:

```
26     foreach (ElementType element in collection) statement
```

27 corresponds to one of two possible expansions:

- 28 • If the `collection` expression is of a type that implements the collection pattern (as defined above), the
 29 expansion of the `foreach` statement is:

```
30     Enumerator enumerator = (collection).GetEnumerator();
31     try {
32         while (enumerator.MoveNext()) {
33             ElementType element = (ElementType)enumerator.Current;
34             statement;
35         }
36     }
37     finally {
38         IDisposable disposable = enumerator as System.IDisposable;
39         if (disposable != null) disposable.Dispose();
40     }
```

41 [Note: Significant optimizations of the above are often easily available. If the type `E` implements
 42 `System.IDisposable`, then the expression `(enumerator as System.IDisposable)` will always
 43 be non-null and the implementation can safely substitute a simple conversion for a possibly more
 44 expensive type test. Conversely, if the type `E` is `sealed` and does not implement
 45 `System.IDisposable`, then the expression `(enumerator as System.IDisposable)` will
 46 always evaluate to null. In this case, the implementation can safely optimize away the entire `finally`
 47 clause. *end note*]

- 48 • Otherwise; the `collection` expression is of a type that implements `System.IEnumerable`, and the
 49 expansion of the `foreach` statement is:

```

1      IEnumerator enumerator =
2      ((System.IEnumerable)(collection)).GetEnumerator();
3      try {
4          while (enumerator.MoveNext()) {
5              ElementType element = (ElementType)enumerator.Current;
6              statement;
7          }
8      }
9      finally {
10         IDisposable disposable = enumerator as System.IDisposable;
11         if (disposable != null) disposable.Dispose();
12     }

```

In either expansion, the `enumerator` variable is a temporary variable that is inaccessible in, and invisible to, the embedded statement, and the `element` variable is read-only in the embedded statement.

[*Example:* The following example prints out each value in a two-dimensional array, in element order:

```

16     using System;
17     class Test
18     {
19         static void Main() {
20             double[,] values = {
21                 {1.2, 2.3, 3.4, 4.5},
22                 {5.6, 6.7, 7.8, 8.9}
23             };
24
25             foreach (double elementValue in values)
26                 Console.Write("{0} ", elementValue);
27             Console.WriteLine();
28         }
29     }

```

The output produced is as follows:

```

31     1.2 2.3 3.4 4.5 5.6 6.7 7.8 8.9

```

32 *end example*]

33 15.9 Jump statements

34 Jump statements unconditionally transfer control.

```

35     jump-statement:
36         break-statement
37         continue-statement
38         goto-statement
39         return-statement
40         throw-statement

```

41 The location to which a jump statement transfers control is called the *target* of the jump statement.

42 When a jump statement occurs within a block, and the target of that jump statement is outside that block, the jump statement is said to *exit* the block. While a jump statement may transfer control out of a block, it can never transfer control into a block.

45 Execution of jump statements is complicated by the presence of intervening `try` statements. In the absence of such `try` statements, a jump statement unconditionally transfers control from the jump statement to its target. In the presence of such intervening `try` statements, execution is more complex. If the jump statement exits one or more `try` blocks with associated `finally` blocks, control is initially transferred to the `finally` block of the innermost `try` statement. When and if control reaches the end point of a `finally` block, control is transferred to the `finally` block of the next enclosing `try` statement. This process is repeated until the `finally` blocks of all intervening `try` statements have been executed.

52 [*Example:* In the example

```

1      using System;
2      class Test
3      {
4          static void Main() {
5              while (true) {
6                  try {
7                      try {
8                          Console.WriteLine("Before break");
9                          break;
10                     }
11                     finally {
12                         Console.WriteLine("Innermost finally block");
13                     }
14                 }
15                 finally {
16                     Console.WriteLine("Outermost finally block");
17                 }
18             }
19             Console.WriteLine("After break");
20         }
21     }

```

22 the finally blocks associated with two try statements are executed before control is transferred to the target
23 of the jump statement.

24 The output produced is as follows:

```

25     Before break
26     Innermost finally block
27     Outermost finally block
28     After break

```

29 *end example]*

30 15.9.1 The break statement

31 The **break** statement exits the nearest enclosing **switch**, **while**, **do**, **for**, or **foreach** statement.

32 *break-statement:*

```
33     break ;
```

34 The target of a **break** statement is the end point of the nearest enclosing **switch**, **while**, **do**, **for**, or
35 **foreach** statement. If a **break** statement is not enclosed by a **switch**, **while**, **do**, **for**, or **foreach**
36 statement, a compile-time error occurs.

37 When multiple **switch**, **while**, **do**, **for**, or **foreach** statements are nested within each other, a **break**
38 statement applies only to the innermost statement. To transfer control across multiple nesting levels, a **goto**
39 statement (§15.9.3) must be used.

40 A **break** statement cannot exit a **finally** block (§15.10). When a **break** statement occurs within a
41 **finally** block, the target of the **break** statement must be within the same **finally** block; otherwise a
42 compile-time error occurs.

43 A **break** statement is executed as follows:

- 44 • If the **break** statement exits one or more **try** blocks with associated **finally** blocks, control is
45 initially transferred to the **finally** block of the innermost **try** statement. When and if control reaches
46 the end point of a **finally** block, control is transferred to the **finally** block of the next enclosing **try**
47 statement. This process is repeated until the **finally** blocks of all intervening **try** statements have
48 been executed.
- 49 • Control is transferred to the target of the **break** statement.

50 Because a **break** statement unconditionally transfers control elsewhere, the end point of a **break** statement
51 is never reachable.

1 15.9.2 The `continue` statement

2 The `continue` statement starts a new iteration of the nearest enclosing `while`, `do`, `for`, or `foreach`
3 statement.

```
4     continue-statement:  
5         continue ;
```

6 The target of a `continue` statement is the end point of the embedded statement of the nearest enclosing
7 `while`, `do`, `for`, or `foreach` statement. If a `continue` statement is not enclosed by a `while`, `do`, `for`, or
8 `foreach` statement, a compile-time error occurs.

9 When multiple `while`, `do`, `for`, or `foreach` statements are nested within each other, a `continue`
10 statement applies only to the innermost statement. To transfer control across multiple nesting levels, a `goto`
11 statement (§15.9.3) must be used.

12 A `continue` statement cannot exit a `finally` block (§15.10). When a `continue` statement occurs within
13 a `finally` block, the target of the `continue` statement must be within the same `finally` block;
14 otherwise a compile-time error occurs.

15 A `continue` statement is executed as follows:

- 16 • If the `continue` statement exits one or more `try` blocks with associated `finally` blocks, control is
17 initially transferred to the `finally` block of the innermost `try` statement. When and if control reaches
18 the end point of a `finally` block, control is transferred to the `finally` block of the next enclosing `try`
19 statement. This process is repeated until the `finally` blocks of all intervening `try` statements have
20 been executed.
- 21 • Control is transferred to the target of the `continue` statement.

22 Because a `continue` statement unconditionally transfers control elsewhere, the end point of a `continue`
23 statement is never reachable.

24 15.9.3 The `goto` statement

25 The `goto` statement transfers control to a statement that is marked by a label.

```
26     goto-statement:  
27         goto identifier ;  
28         goto case constant-expression ;  
29         goto default ;
```

30 The target of a `goto identifier` statement is the labeled statement with the given label. If a label with the
31 given name does not exist in the current function member, or if the `goto` statement is not within the scope of
32 the label, a compile-time error occurs. [Note: This rule permits the use of a `goto` statement to transfer
33 control *out of* a nested scope, but not *into* a nested scope. In the example

```
34     using System;  
35     class Test  
36     {  
37         static void Main(string[] args) {  
38             string[,] table = {  
39                 {"red", "blue", "green"},  
40                 {"Monday", "Wednesday", "Friday"}  
41             };  
42             foreach (string str in args) {  
43                 int row, colm;  
44                 for (row = 0; row <= 1; ++row)  
45                     for (colm = 0; colm <= 2; ++colm)  
46                         if (str == table[row,colm])  
47                             goto done;
```

```

1         Console.WriteLine("{0} not found", str);
2         continue;
3     done:
4         Console.WriteLine("Found {0} at [{1}][{2}]", str, row, col);
5     }
6 }
7 }

```

8 a `goto` statement is used to transfer control out of a nested scope. *end note*]

9 The target of a `goto case` statement is the statement list in the immediately enclosing `switch` statement (§15.7.2) which contains a `case` label with the given constant value. If the `goto case` statement is not enclosed by a `switch` statement, if the *constant-expression* is not implicitly convertible (§13.1) to the governing type of the nearest enclosing `switch` statement, or if the nearest enclosing `switch` statement does not contain a `case` label with the given constant value, a compile-time error occurs.

14 The target of a `goto default` statement is the statement list in the immediately enclosing `switch` statement (§15.7.2), which contains a `default` label. If the `goto default` statement is not enclosed by a `switch` statement, or if the nearest enclosing `switch` statement does not contain a `default` label, a compile-time error occurs.

18 A `goto` statement cannot exit a `finally` block (§15.10). When a `goto` statement occurs within a `finally` block, the target of the `goto` statement must be within the same `finally` block, or otherwise a compile-time error occurs.

21 A `goto` statement is executed as follows:

- 22 • If the `goto` statement exits one or more `try` blocks with associated `finally` blocks, control is initially transferred to the `finally` block of the innermost `try` statement. When and if control reaches the end point of a `finally` block, control is transferred to the `finally` block of the next enclosing `try` statement. This process is repeated until the `finally` blocks of all intervening `try` statements have been executed.
- 27 • Control is transferred to the target of the `goto` statement.

28 Because a `goto` statement unconditionally transfers control elsewhere, the end point of a `goto` statement is never reachable.

30 15.9.4 The return statement

31 The `return` statement returns control to the caller of the function member in which the `return` statement appears.

```

33     return-statement:
34     return expressionopt ;

```

35 A `return` statement with no expression can be used only in a function member that does not compute a value; that is, a method with the return type `void`, the `set` accessor of a property or indexer, the add and remove accessors of an event, an instance constructor, static constructor, or a destructor.

38 A `return` statement with an expression can only be used in a function member that computes a value, that is, a method with a non-void return type, the `get` accessor of a property or indexer, or a user-defined operator. An implicit conversion (§13.1) must exist from the type of the expression to the return type of the containing function member.

42 It is a compile-time error for a `return` statement to appear in a `finally` block (§15.10).

43 A `return` statement is executed as follows:

- 44 • If the `return` statement specifies an expression, the expression is evaluated and the resulting value is converted to the return type of the containing function member by an implicit conversion. The result of the conversion becomes the value returned to the caller.

- 1 • If the `return` statement is enclosed by one or more `try` blocks with associated `finally` blocks,
2 control is initially transferred to the `finally` block of the innermost `try` statement. When and if
3 control reaches the end point of a `finally` block, control is transferred to the `finally` block of the
4 next enclosing `try` statement. This process is repeated until the `finally` blocks of all enclosing `try`
5 statements have been executed.
- 6 • Control is returned to the caller of the containing function member.

7 Because a `return` statement unconditionally transfers control elsewhere, the end point of a `return`
8 statement is never reachable.

9 **15.9.5 The `throw` statement**

10 The `throw` statement throws an exception.

```
11 throw-statement:  
12 throw expressionopt ;
```

13 A `throw` statement with an expression throws the value produced by evaluating the expression. The
14 expression must denote a value of the class type `System.Exception` or of a class type that derives from
15 `System.Exception`. If evaluation of the expression produces `null`, a
16 `System.NullReferenceException` is thrown instead.

17 A `throw` statement with no expression can be used only in a `catch` block, in which case, that statement re-
18 throws the exception that is currently being handled by that `catch` block.

19 Because a `throw` statement unconditionally transfers control elsewhere, the end point of a `throw` statement
20 is never reachable.

21 When an exception is thrown, control is transferred to the first `catch` clause in an enclosing `try` statement
22 that can handle the exception. The process that takes place from the point of the exception being thrown to
23 the point of transferring control to a suitable exception handler is known as *exception propagation*.
24 Propagation of an exception consists of repeatedly evaluating the following steps until a `catch` clause that
25 matches the exception is found. In this description, the *throw point* is initially the location at which the
26 exception is thrown.

- 27 • In the current function member, each `try` statement that encloses the throw point is examined. For each
28 statement `S`, starting with the innermost `try` statement and ending with the outermost `try` statement, the
29 following steps are evaluated:
 - 30 ○ If the `try` block of `S` encloses the throw point and if `S` has one or more `catch` clauses, the `catch`
31 clauses are examined in order of appearance to locate a suitable handler for the exception. The first
32 `catch` clause that specifies the exception type or a base type of the exception type is considered a
33 match. A general `catch` (§15.10) clause is considered a match for any exception type. If a matching
34 `catch` clause is located, the exception propagation is completed by transferring control to the block
35 of that `catch` clause.
 - 36 ○ Otherwise, if the `try` block or a `catch` block of `S` encloses the throw point and if `S` has a `finally`
37 block, control is transferred to the `finally` block. If the `finally` block throws another exception,
38 processing of the current exception is terminated. Otherwise, when control reaches the end point of
39 the `finally` block, processing of the current exception is continued.
- 40 • If an exception handler was not located in the current function member invocation, the function member
41 invocation is terminated. The steps above are then repeated for the caller of the function member with a
42 throw point corresponding to the statement from which the function member was invoked.
- 43 • If the exception processing terminates all function member invocations in the current thread, indicating
44 that the thread has no handler for the exception, then the thread is itself terminated. The impact of such
45 termination is implementation-defined.

1 15.10 The try statement

2 The `try` statement provides a mechanism for catching exceptions that occur during execution of a block.
3 Furthermore, the `try` statement provides the ability to specify a block of code that is always executed when
4 control leaves the `try` statement.

```
5     try-statement:
6         try block catch-clauses
7         try block finally-clause
8         try block catch-clauses finally-clause

9     catch-clauses:
10        specific-catch-clauses general-catch-clauseopt
11        specific-catch-clausesopt general-catch-clause

12    specific-catch-clauses:
13        specific-catch-clause
14        specific-catch-clauses specific-catch-clause

15    specific-catch-clause:
16        catch ( class-type identifieropt ) block

17    general-catch-clause:
18        catch block

19    finally-clause:
20        finally block
```

21 There are three possible forms of `try` statements:

- 22 • A `try` block followed by one or more `catch` blocks.
- 23 • A `try` block followed by a `finally` block.
- 24 • A `try` block followed by one or more `catch` blocks followed by a `finally` block.

25 When a `catch` clause specifies a *class-type*, the type must be `System.Exception` or a type that derives
26 from `System.Exception`.

27 When a `catch` clause specifies both a *class-type* and an *identifier*, an **exception variable** of the given name
28 and type is declared. The exception variable corresponds to a local variable with a scope that extends over
29 the `catch` block. During execution of the `catch` block, the exception variable represents the exception
30 currently being handled. For purposes of definite assignment checking, the exception variable is considered
31 definitely assigned in its entire scope.

32 Unless a `catch` clause includes an exception variable name, it is impossible to access the exception object
33 in the `catch` block.

34 A `catch` clause that specifies neither an exception type nor an exception variable name is called a general
35 `catch` clause. A `try` statement can only have one general `catch` clause, and if one is present it must be the
36 last `catch` clause.

37 [*Note:* Some environments, especially those supporting multiple languages, may support exceptions that are
38 not representable as an object derived from `System.Exception`, although such an exception could never
39 be generated by C# code. In such an environment, a general `catch` clause might be used to catch such an
40 exception. Thus, a general `catch` clause is semantically different from one that specifies the type
41 `System.Exception`, in that the former may also catch exceptions from other languages. *end note*]

42 In order to locate a handler for an exception, `catch` clauses are examined in lexical order. A compile-time
43 error occurs if a `catch` clause specifies a type that is the same as, or is derived from, a type that was
44 specified in an earlier `catch` clause for the same `try`. [*Note:* Without this restriction, it would be possible to
45 write unreachable `catch` clauses. *end note*]

1 Within a `catch` block, a `throw` statement (§15.9.5) with no expression can be used to re-throw the
 2 exception that was caught by the `catch` block. Assignments to an exception variable do not alter the
 3 exception that is re-thrown.

4 [*Example:* In the example

```

5     using System;
6     class Test
7     {
8         static void F() {
9             try {
10                G();
11            }
12            catch (Exception e) {
13                Console.WriteLine("Exception in F: " + e.Message);
14                e = new Exception("F");
15                throw;           // re-throw
16            }
17        }
18
19        static void G() {
20            throw new Exception("G");
21        }
22
23        static void Main() {
24            try {
25                F();
26            }
27            catch (Exception e) {
28                Console.WriteLine("Exception in Main: " + e.Message);
29            }
30        }
31    }

```

30 the method `F` catches an exception, writes some diagnostic information to the console, alters the exception
 31 variable, and re-throws the exception. The exception that is re-thrown is the original exception, so the output
 32 produced is:

```

33     Exception in F: G
34     Exception in Main: G

```

35 If the first `catch` block had thrown `e` instead of rethrowing the current exception, the output produced would
 36 be as follows:

```

37     Exception in F: G
38     Exception in Main: F

```

39 *end example]*

40 It is a compile-time error for a `break`, `continue`, or `goto` statement to transfer control out of a `finally`
 41 block. When a `break`, `continue`, or `goto` statement occurs in a `finally` block, the target of the statement
 42 must be within the same `finally` block, or otherwise a compile-time error occurs.

43 It is a compile-time error for a `return` statement to occur in a `finally` block.

44 A `try` statement is executed as follows:

- 45 • Control is transferred to the `try` block.
- 46 • When and if control reaches the end point of the `try` block:
 - 47 ○ If the `try` statement has a `finally` block, the `finally` block is executed.
 - 48 ○ Control is transferred to the end point of the `try` statement.
- 49 • If an exception is propagated to the `try` statement during execution of the `try` block:
 - 50 ○ The `catch` clauses, if any, are examined in order of appearance to locate a suitable handler for the
 51 exception. The first `catch` clause that specifies the exception type or a base type of the exception

1 type is considered a match. A general `catch` clause is considered a match for any exception type. If
 2 a matching `catch` clause is located:

- 3 • If the matching `catch` clause declares an exception variable, the exception object is assigned to
 4 the exception variable.
- 5 • Control is transferred to the matching `catch` block.
- 6 • When and if control reaches the end point of the `catch` block:
 - 7 ○ If the `try` statement has a `finally` block, the `finally` block is executed.
 - 8 ○ Control is transferred to the end point of the `try` statement.
- 9 • If an exception is propagated to the `try` statement during execution of the `catch` block:
 - 10 ○ If the `try` statement has a `finally` block, the `finally` block is executed.
 - 11 ○ The exception is propagated to the next enclosing `try` statement.
- 12 ○ If the `try` statement has no `catch` clauses or if no `catch` clause matches the exception:
 - 13 • If the `try` statement has a `finally` block, the `finally` block is executed.
 - 14 • The exception is propagated to the next enclosing `try` statement.

15 The statements of a `finally` block are always executed when control leaves a `try` statement. This is true
 16 whether the control transfer occurs as a result of normal execution, as a result of executing a `break`,
 17 `continue`, `goto`, or `return` statement, or as a result of propagating an exception out of the `try` statement.

18 If an exception is thrown during execution of a `finally` block, the exception is propagated to the next
 19 enclosing `try` statement. If another exception was in the process of being propagated, that exception is lost.
 20 The process of propagating an exception is discussed further in the description of the `throw` statement
 21 (§15.9.5).

22 The `try` block of a `try` statement is reachable if the `try` statement is reachable.

23 A `catch` block of a `try` statement is reachable if the `try` statement is reachable.

24 The `finally` block of a `try` statement is reachable if the `try` statement is reachable.

25 The end point of a `try` statement is reachable if both of the following are true:

- 26 • The end point of the `try` block is reachable or the end point of at least one `catch` block is reachable.
- 27 • If a `finally` block is present, the end point of the `finally` block is reachable.

28 15.11 The checked and unchecked statements

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

31 *checked-statement:*
 32 `checked block`

33 *unchecked-statement:*
 34 `unchecked block`

35 The `checked` statement causes all expressions in the *block* to be evaluated in a checked context, and the
 36 `unchecked` statement causes all expressions in the *block* to be evaluated in an unchecked context.

37 The `checked` and `unchecked` statements are precisely equivalent to the `checked` and `unchecked`
 38 operators (§14.5.12), except that they operate on blocks instead of expressions.

1 15.12 The lock statement

2 The `lock` statement obtains the mutual-exclusion lock for a given object, executes a statement, and then
3 releases the lock.

4 *lock-statement:*
5 `lock (expression) embedded-statement`

6 The expression of a `lock` statement must denote a value of a *reference-type*. No implicit boxing conversion
7 (§13.1.5) is ever performed for the expression of a `lock` statement, and thus it is a compile-time error for the
8 expression to denote a value of a *value-type*.

9 A `lock` statement of the form

10 `lock (x) ...`

11 where `x` is an expression of a *reference-type*, is precisely equivalent to

```
12     System.Threading.Monitor.Enter(x);
13     try {
14         ...
15     }
16     finally {
17         System.Threading.Monitor.Exit(x);
18     }
```

19 except that `x` is only evaluated once.

20 [*Example:* The `System.Type` object of a class can conveniently be used as the mutual-exclusion lock for
21 static methods of the class. For example:

```
22     class Cache
23     {
24         public static void Add(object x) {
25             lock (typeof(Cache)) {
26                 ...
27             }
28         }
29         public static void Remove(object x) {
30             lock (typeof(Cache)) {
31                 ...
32             }
33         }
34     }
```

35 *end example]*

36 15.13 The using statement

37 The `using` statement obtains one or more resources, executes a statement, and then disposes of the resource.

38 *using-statement:*
39 `using (resource-acquisition) embedded-statement`

40 *resource-acquisition:*
41 `local-variable-declaration`
42 `expression`

43 A *resource* is a class or struct that implements `System.IDisposable`, which includes a single
44 parameterless method named `Dispose`. Code that is using a resource can call `Dispose` to indicate that the
45 resource is no longer needed. If `Dispose` is not called, then automatic disposal eventually occurs as a
46 consequence of garbage collection.

47 If the form of *resource-acquisition* is *local-variable-declaration* then the type of the *local-variable-*
48 *declaration* must be `System.IDisposable` or a type that can be implicitly converted to
49 `System.IDisposable`. If the form of *resource-acquisition* is *expression* then this expression must be
50 `System.IDisposable` or a type that can be implicitly converted to `System.IDisposable`.

1 Local variables declared in a *resource-acquisition* are read-only, and must include an initializer. A compile-
 2 time error occurs if the embedded statement attempts to modify these local variables (via assignment or the
 3 ++ and -- operators) or pass them as *ref* or *out* parameters.

4 A *using* statement is translated into three parts: acquisition, usage, and disposal. Usage of the resource is
 5 implicitly enclosed in a *try* statement that includes a *finally* clause. This *finally* clause disposes of the
 6 resource. If a *null* resource is acquired, then no call to *Dispose* is made, and no exception is thrown.

7 A *using* statement of the form

```
8     using (R r1 = new R()) {
9         r1.F();
10    }
```

11 is precisely equivalent to

```
12     R r1 = new R();
13     try {
14         r1.F();
15     }
16     finally {
17         if (r1 != null) ((IDisposable)r1).Dispose();
18     }
```

19 A *resource-acquisition* may acquire multiple resources of a given type. This is equivalent to nested *using*
 20 statements. A *using* statement of the form

```
21     using (R r1 = new R(), r2 = new R()) {
22         r1.F();
23         r2.F();
24     }
```

25 is precisely equivalent to:

```
26     using (R r1 = new R())
27         using (R r2 = new R()) {
28         r1.F();
29         r2.F();
30     }
```

31 which is, by expansion, precisely equivalent to:

```
32     R r1 = new R();
33     try {
34         R r2 = new R();
35         try {
36             r1.F();
37             r2.F();
38         }
39         finally {
40             if (r2 != null) ((IDisposable)r2).Dispose();
41         }
42     }
43     finally {
44         if (r1 != null) ((IDisposable)r1).Dispose();
45     }
```


16. Namespaces

1

2 C# programs are organized using namespaces. Namespaces are used both as an “internal” organization
3 system for a program, and as an “external” organization system—a way of presenting program elements that
4 are exposed to other programs.

5 Using-directives (§16.3) are provided to facilitate the use of namespaces.

6 16.1 Compilation units

7 A *compilation-unit* defines the overall structure of a source file. A compilation unit consists of zero or more
8 *using-directives* followed by zero or more *global-attributes* followed by zero or more *namespace-member-*
9 *declarations*.

10 *compilation-unit:*

11 *using-directives*_{opt} *global-attributes*_{opt} *namespace-member-declarations*_{opt}

12 A C# program consists of one or more compilation units, each contained in a separate source file. When a
13 C# program is compiled, all of the compilation units are processed together. Thus, compilation units can
14 depend on each other, possibly in a circular fashion.

15 The *using-directives* of a compilation unit affect the *global-attributes* and *namespace-member-declarations*
16 of that compilation unit, but have no effect on other compilation units.

17 The *global-attributes* (§24) of a compilation unit permit the specification of attributes for the target
18 assembly. Assemblies act as physical containers for types.

19 The *namespace-member-declarations* of each compilation unit of a program contribute members to a single
20 declaration space called the global namespace. [*Example:* For example:

21 File A.cs:

22 class A {}

23 File B.cs:

24 class B {}

25 The two compilation units contribute to the single global namespace, in this case declaring two classes with
26 the fully qualified names A and B. Because the two compilation units contribute to the same declaration
27 space, it would have been an error if each contained a declaration of a member with the same name. *end*
28 *example*]

29 16.2 Namespace declarations

30 A *namespace-declaration* consists of the keyword `namespace`, followed by a namespace name and body,
31 optionally followed by a semicolon.

32 *namespace-declaration:*

33 namespace *qualified-identifier* *namespace-body* ;_{opt}

34 *qualified-identifier:*

35 *identifier*

36 *qualified-identifier* . *identifier*

37 *namespace-body:*

38 { *using-directives*_{opt} *namespace-member-declarations*_{opt} }

1 A *namespace-declaration* may occur as a top-level declaration in a *compilation-unit* or as a member
 2 declaration within another *namespace-declaration*. When a *namespace-declaration* occurs as a top-level
 3 declaration in a *compilation-unit*, the namespace becomes a member of the global namespace. When a
 4 *namespace-declaration* occurs within another *namespace-declaration*, the inner namespace becomes a
 5 member of the outer namespace. In either case, the name of a namespace must be unique within the
 6 containing namespace.

7 Namespaces are implicitly `public` and the declaration of a namespace cannot include any access modifiers.

8 Within a *namespace-body*, the optional *using-directives* import the names of other namespaces and types,
 9 allowing them to be referenced directly instead of through qualified names. The optional *namespace-*
 10 *member-declarations* contribute members to the declaration space of the namespace. Note that all *using-*
 11 *directives* must appear before any member declarations.

12 The *qualified-identifier* of a *namespace-declaration* may be a single identifier or a sequence of identifiers
 13 separated by “.” tokens. The latter form permits a program to define a nested namespace without lexically
 14 nesting several namespace declarations. [*Example:* For example,

```
15     namespace N1.N2
16     {
17         class A {}
18         class B {}
19     }
```

20 is semantically equivalent to

```
21     namespace N1
22     {
23         namespace N2
24         {
25             class A {}
26             class B {}
27         }
28     }
```

29 *end example*]

30 Namespaces are open-ended, and two namespace declarations with the same fully qualified name contribute
 31 to the same declaration space (§10.3). [*Example:* In the example

```
32     namespace N1.N2
33     {
34         class A {}
35     }
36     namespace N1.N2
37     {
38         class B {}
39     }
```

40 the two namespace declarations above contribute to the same declaration space, in this case declaring two
 41 classes with the fully qualified names `N1.N2.A` and `N1.N2.B`. Because the two declarations contribute to
 42 the same declaration space, it would have been an error if each contained a declaration of a member with the
 43 same name. *end example*]

44 16.3 Using directives

45 *Using-directives* facilitate the use of namespaces and types defined in other namespaces. *Using-directives*
 46 impact the name resolution process of *namespace-or-type-names* (§10.8) and *simple-names* (§14.5.2), but
 47 unlike declarations, *using-directives* do not contribute new members to the underlying declaration spaces of
 48 the compilation units or namespaces within which they are used.

```
49     using-directives:
50         using-directive
51         using-directives using-directive
```



```

1      using-directive:
2          using-alias-directive
3          using-namespace-directive

```

4 A *using-alias-directive* (§16.3.1) introduces an alias for a namespace or type.

5 A *using-namespace-directive* (§16.3.2) imports the type members of a namespace.

6 The scope of a *using-directive* extends over the *namespace-member-declarations* of its immediately
7 containing compilation unit or namespace body. The scope of a *using-directive* specifically does not include
8 its peer *using-directives*. Thus, peer *using-directives* do not affect each other, and the order in which they are
9 written is insignificant.

10 16.3.1 Using alias directives

11 A *using-alias-directive* introduces an identifier that serves as an alias for a namespace or type within the
12 immediately enclosing compilation unit or namespace body.

```

13      using-alias-directive:
14          using identifier = namespace-or-type-name ;

```

15 Within member declarations in a compilation unit or namespace body that contains a *using-alias-directive*,
16 the identifier introduced by the *using-alias-directive* can be used to reference the given namespace or type.

17 [Example: For example:

```

18      namespace N1.N2
19      {
20          class A {}
21      }
22      namespace N3
23      {
24          using A = N1.N2.A;
25          class B: A {}
26      }

```

27 Above, within member declarations in the N3 namespace, A is an alias for N1.N2.A, and thus class N3.B
28 derives from class N1.N2.A. The same effect can be obtained by creating an alias R for N1.N2 and then
29 referencing R.A:

```

30      namespace N3
31      {
32          using R = N1.N2;
33          class B: R.A {}
34      }

```

35 *end example]*

36 The *identifier* of a *using-alias-directive* must be unique within the declaration space of the compilation unit
37 or namespace that immediately contains the *using-alias-directive*. [Example: For example:

```

38      namespace N3
39      {
40          class A {}
41      }
42      namespace N3
43      {
44          using A = N1.N2.A;      // Error, A already exists
45      }

```

46 Above, N3 already contains a member A, so it is a compile-time error for a *using-alias-directive* to use that
47 identifier. *end example]* Likewise, it is a compile-time error for two or more *using-alias-directives* in the
48 same compilation unit or namespace body to declare aliases by the same name.

49 A *using-alias-directive* makes an alias available within a particular compilation unit or namespace body, but
50 it does not contribute any new members to the underlying declaration space. In other words, a *using-alias-*

C# LANGUAGE SPECIFICATION

1 *directive* is not transitive, but, rather, affects only the compilation unit or namespace body in which it occurs.
2 [Example: In the example

```
3     namespace N3
4     {
5         using R = N1.N2;
6     }
7     namespace N3
8     {
9         class B: R.A {}           // Error, R unknown
10    }
```

11 the scope of the *using-alias-directive* that introduces R only extends to member declarations in the
12 namespace body in which it is contained, so R is unknown in the second namespace declaration. However,
13 placing the *using-alias-directive* in the containing compilation unit causes the alias to become available
14 within both namespace declarations:

```
15     using R = N1.N2;
16     namespace N3
17     {
18         class B: R.A {}
19     }
20     namespace N3
21     {
22         class C: R.A {}
23     }
```

24 *end example*]

25 Just like regular members, names introduced by *using-alias-directives* are hidden by similarly named
26 members in nested scopes. [Example: In the example

```
27     using R = N1.N2;
28     namespace N3
29     {
30         class R {}
31         class B: R.A {}           // Error, R has no member A
32     }
```

33 the reference to R.A in the declaration of B causes a compile-time error because R refers to N3.R, not
34 N1.N2. *end example*]

35 The order in which *using-alias-directives* are written has no significance, and resolution of the *namespace-*
36 *or-type-name* referenced by a *using-alias-directive* is not affected by the *using-alias-directive* itself or by
37 other *using-directives* in the immediately containing compilation unit or namespace body. In other words,
38 the *namespace-or-type-name* of a *using-alias-directive* is resolved as if the immediately containing
39 compilation unit or namespace body had no *using-directives*. [Example: In the example

```
40     namespace N1.N2 {}
41     namespace N3
42     {
43         using R1 = N1;           // OK
44         using R2 = N1.N2;       // OK
45         using R3 = R1.N2;       // Error, R1 unknown
46     }
```

47 the last *using-alias-directive* results in a compile-time error because it is not affected by the first *using-alias-*
48 *directive*. *end example*]

49 A *using-alias-directive* can create an alias for any namespace or type, including the namespace within which
50 it appears and any namespace or type nested within that namespace.

1 Accessing a namespace or type through an alias yields exactly the same result as accessing that namespace
2 or type through its declared name. [*Example:* For example, given

```

3     namespace N1.N2
4     {
5         class A {}
6     }
7     namespace N3
8     {
9         using R1 = N1;
10        using R2 = N1.N2;
11
12        class B
13        {
14            N1.N2.A a;           // refers to N1.N2.A
15            R1.N2.A b;          // refers to N1.N2.A
16            R2.A c;             // refers to N1.N2.A
17        }

```

18 the names `N1.N2.A`, `R1.N2.A`, and `R2.A` are equivalent and all refer to the class whose fully qualified
19 name is `N1.N2.A`. *end example*]

20 16.3.2 Using namespace directives

21 A *using-namespace-directive* imports the types contained in a namespace into the immediately enclosing
22 compilation unit or namespace body, enabling the identifier of each type to be used without qualification.

```

23     using-namespace-directive:
24     using namespace-name ;

```

25 Within member declarations in a compilation unit or namespace body that contains a *using-namespace-*
26 *directive*, the types contained in the given namespace can be referenced directly. [*Example:* For example:

```

27     namespace N1.N2
28     {
29         class A {}
30     }
31     namespace N3
32     {
33         using N1.N2;
34         class B: A {}
35     }

```

36 Above, within member declarations in the `N3` namespace, the type members of `N1.N2` are directly
37 available, and thus class `N3.B` derives from class `N1.N2.A`. *end example*]

38 A *using-namespace-directive* imports the types contained in the given namespace, but specifically does not
39 import nested namespaces. [*Example:* In the example

```

40     namespace N1.N2
41     {
42         class A {}
43     }
44     namespace N3
45     {
46         using N1;
47         class B: N2.A {}      // Error, N2 unknown
48     }

```

49 the *using-namespace-directive* imports the types contained in `N1`, but not the namespaces nested in `N1`. Thus,
50 the reference to `N2.A` in the declaration of `B` results in a compile-time error because no members named `N2`
51 are in scope. *end example*]

52 Unlike a *using-alias-directive*, a *using-namespace-directive* may import types whose identifiers are already
53 defined within the enclosing compilation unit or namespace body. In effect, names imported by a *using-*

C# LANGUAGE SPECIFICATION

1 *namespace-directive* are hidden by similarly named members in the enclosing compilation unit or
2 namespace body. [*Example*: For example:

```
3     namespace N1.N2
4     {
5         class A {}
6         class B {}
7     }
8     namespace N3
9     {
10        using N1.N2;
11        class A {}
12    }
```

13 Here, within member declarations in the N3 namespace, A refers to N3.A rather than N1.N2.A. *end example*]

14 When more than one namespace imported by *using-namespace-directives* in the same compilation unit or
15 namespace body contain types by the same name, references to that name are considered ambiguous.

16 [*Example*: In the example

```
17     namespace N1
18     {
19         class A {}
20     }
21     namespace N2
22     {
23         class A {}
24     }
25     namespace N3
26     {
27         using N1;
28         using N2;
29         class B: A {}           // Error, A is ambiguous
30     }
```

31 both N1 and N2 contain a member A, and because N3 imports both, referencing A in N3 is a compile-time
32 error. *end example*] In this situation, the conflict can be resolved either through qualification of references
33 to A, or by introducing a *using-alias-directive* that picks a particular A. [*Example*: For example:

```
34     namespace N3
35     {
36         using N1;
37         using N2;
38         using A = N1.A;
39         class B: A {}           // A means N1.A
40     }
```

41 *end example*]

42 Like a *using-alias-directive*, a *using-namespace-directive* does not contribute any new members to the
43 underlying declaration space of the compilation unit or namespace, but, rather, affects only the compilation
44 unit or namespace body in which it appears.

45 The *namespace-name* referenced by a *using-namespace-directive* is resolved in the same way as the
46 *namespace-or-type-name* referenced by a *using-alias-directive*. Thus, *using-namespace-directives* in the
47 same compilation unit or namespace body do not affect each other and can be written in any order.

48 16.4 Namespace members

49 A *namespace-member-declaration* is either a *namespace-declaration* (§16.2) or a *type-declaration* (§16.5).

1 *namespace-member-declarations:*
 2 *namespace-member-declaration*
 3 *namespace-member-declarations namespace-member-declaration*

4 *namespace-member-declaration:*
 5 *namespace-declaration*
 6 *type-declaration*

7 A compilation unit or a namespace body can contain *namespace-member-declarations*, and such
 8 declarations contribute new members to the underlying declaration space of the containing compilation unit
 9 or namespace body.

10 **16.5 Type declarations**

11 A *type-declaration* is a *class-declaration* (§17.1), a *struct-declaration* (§18.1), an *interface-declaration*
 12 (§20.1), an *enum-declaration* (§21.1), or a *delegate-declaration* (§22.1).

13 *type-declaration:*
 14 *class-declaration*
 15 *struct-declaration*
 16 *interface-declaration*
 17 *enum-declaration*
 18 *delegate-declaration*

19 A *type-declaration* can occur as a top-level declaration in a compilation unit or as a member declaration
 20 within a namespace, class, or struct.

21 When a type declaration for a type *T* occurs as a top-level declaration in a compilation unit, the fully
 22 qualified name of the newly declared type is simply *T*. When a type declaration for a type *T* occurs within a
 23 namespace, class, or struct, the fully qualified name of the newly declared type is *N.T*, where *N* is the fully
 24 qualified name of the containing namespace, class, or struct.

25 A type declared within a class or struct is called a nested type (§17.2.6).

26 The permitted access modifiers and the default access for a type declaration depend on the context in which
 27 the declaration takes place (§10.5.1):

- 28 • Types declared in compilation units or namespaces can have `public` or `internal` access. The default
 29 is `internal` access.
- 30 • Types declared in classes can have `public`, `protected internal`, `protected`, `internal`, or
 31 `private` access. The default is `private` access.
- 32 • Types declared in structs can have `public`, `internal`, or `private` access. The default is `private`
 33 access.

17. Classes

1

2 A class is a data structure that may contain data members (constants and fields), function members
 3 (methods, properties, events, indexers, operators, instance constructors, destructors, and static constructors),
 4 and nested types. Class types support inheritance, a mechanism whereby a *derived class* can extend and
 5 specialize a *base class*.

6 17.1 Class declarations

7 A *class-declaration* is a *type-declaration* (§16.5) that declares a new class.

8 *class-declaration:*

9 *attributes*_{opt} *class-modifiers*_{opt} **class** *identifier* *class-base*_{opt} *class-body* ;_{opt}

10 A *class-declaration* consists of an optional set of *attributes* (§24), followed by an optional set of *class-*
 11 *modifiers* (§17.1.1), followed by the keyword **class** and an *identifier* that names the class, followed by an
 12 optional *class-base* specification (§17.1.2), followed by a *class-body* (§17.1.3), optionally followed by a
 13 semicolon.

14 17.1.1 Class modifiers

15 A *class-declaration* may optionally include a sequence of class modifiers:

16 *class-modifiers:*

17 *class-modifier*

18 *class-modifiers* *class-modifier*

19 *class-modifier:*

20 **new**

21 **public**

22 **protected**

23 **internal**

24 **private**

25 **abstract**

26 **sealed**

27 It is a compile-time error for the same modifier to appear multiple times in a class declaration.

28 The **new** modifier is permitted on nested classes. It specifies that the class hides an inherited member by the
 29 same name, as described in §10.2.2. It is a compile-time error for the **new** modifier to appear on a class
 30 declaration that is not a nested class declaration.

31 The **public**, **protected**, **internal**, and **private** modifiers control the accessibility of the class.
 32 Depending on the context in which the class declaration occurs, some of these modifiers may not be
 33 permitted (§10.5.1).

34 The **abstract** and **sealed** modifiers are discussed in the following sections.

35 17.1.1.1 Abstract classes

36 The **abstract** modifier is used to indicate that a class is incomplete and that it is intended to be used only
 37 as a base class. An *abstract class* differs from a *non-abstract class* in the following ways:

- 38 • An abstract class cannot be instantiated directly, and it is a compile-time error to use the **new** operator on
 39 an abstract class. While it is possible to have variables and values whose compile-time types are
 40 abstract, such variables and values will necessarily either be `null` or contain references to instances of
 41 non-abstract classes derived from the abstract types.

- 1 • An abstract class is permitted (but not required) to contain abstract members.
- 2 • An abstract class cannot be sealed.

3 When a non-abstract class is derived from an abstract class, the non-abstract class must include actual
 4 implementations of all inherited abstract members, thereby overriding those abstract members. [*Example*: In
 5 the example

```

6     abstract class A
7     {
8         public abstract void F();
9     }
10    abstract class B: A
11    {
12        public void G() {}
13    }
14    class C: B
15    {
16        public override void F() {
17            // actual implementation of F
18        }
19    }
  
```

20 the abstract class A introduces an abstract method F. Class B introduces an additional method G, but since it
 21 doesn't provide an implementation of F, B must also be declared abstract. Class C overrides F and provides
 22 an actual implementation. Since there are no abstract members in C, C is permitted (but not required) to be
 23 non-abstract. *end example*]

24 17.1.1.2 Sealed classes

25 The `sealed` modifier is used to prevent derivation from a class. A compile-time error occurs if a sealed
 26 class is specified as the base class of another class.

27 A sealed class cannot also be an abstract class.

28 [*Note*: The `sealed` modifier is primarily used to prevent unintended derivation, but it also enables certain
 29 run-time optimizations. In particular, because a sealed class is known to never have any derived classes, it is
 30 possible to transform virtual function member invocations on sealed class instances into non-virtual
 31 invocations. *end note*]

32 17.1.2 Class base specification

33 A class declaration may include a *class-base* specification, which defines the direct base class of the class
 34 and the interfaces (§20) implemented by the class.

```

35     class-base:
36         : class-type
37         : interface-type-list
38         : class-type , interface-type-list
39     interface-type-list:
40         interface-type
41         interface-type-list , interface-type
  
```

42 17.1.2.1 Base classes

43 When a *class-type* is included in the *class-base*, it specifies the direct base class of the class being declared.
 44 If a class declaration has no *class-base*, or if the *class-base* lists only interface types, the direct base class is
 45 assumed to be `object`. A class inherits members from its direct base class, as described in §17.2.1.

46 [*Example*: In the example

```

47     class A {}
48     class B: A {}
  
```


1 class A is said to be the direct base class of B, and B is said to be derived from A. Since A does not explicitly
2 specify a direct base class, its direct base class is implicitly `object`. *end example*]

3 The direct base class of a class type must be at least as accessible as the class type itself (§10.5.4). For
4 example, it is a compile-time error for a `public` class to derive from a `private` or `internal` class.

5 The direct base class of a class type must not be any of the following types: `System.Array`,
6 `System.Delegate`, `System.Enum`, or `System.ValueType`.

7 The base classes of a class are the direct base class and its base classes. In other words, the set of base
8 classes is the transitive closure of the direct base class relationship. [*Note*: Referring to the example above,
9 the base classes of B are A and `object`. *end note*]

10 Except for class `object`, every class has exactly one direct base class. The `object` class has no direct base
11 class and is the ultimate base class of all other classes.

12 When a class B derives from a class A, it is a compile-time error for A to depend on B. A class *directly*
13 *depends on* its direct base class (if any) and *directly depends on* the class within which it is immediately
14 nested (if any). Given this definition, the complete set of classes upon which a class depends is the transitive
15 closure of the *directly depends on* relationship.

16 [*Example*: The example

```
17     class A: B {}
18     class B: C {}
19     class C: A {}
```

20 is in error because the classes circularly depend on themselves. Likewise, the example

```
21     class A: B.C {}
22     class B: A
23     {
24     public class C {}
25     }
```

26 results in a compile-time error because A depends on B.C (its direct base class), which depends on B (its
27 immediately enclosing class), which circularly depends on A. *end example*]

28 Note that a class does not depend on the classes that are nested within it. [*Example*: In the example

```
29     class A
30     {
31     class B: A {}
32     }
```

33 B depends on A (because A is both its direct base class and its immediately enclosing class), but A does not
34 depend on B (since B is neither a base class nor an enclosing class of A). Thus, the example is valid. *end*
35 *example*]

36 It is not possible to derive from a `sealed` class. [*Example*: In the example

```
37     sealed class A {}
38     class B: A {}           // Error, cannot derive from a sealed class
```

39 class B results in a compile-time error because it attempts to derive from the `sealed` class A. *end example*]

40 17.1.2.2 Interface implementations

41 A *class-base* specification may include a list of interface types, in which case the class is said to implement
42 the given interface types. Interface implementations are discussed further in §20.4.

43 17.1.3 Class body

44 The *class-body* of a class defines the members of that class.

```

1      class-body:
2          { class-member-declarationsopt }

```

3 17.2 Class members

4 The members of a class consist of the members introduced by its *class-member-declarations* and the
5 members inherited from the direct base class.

```

6      class-member-declarations:
7          class-member-declaration
8          class-member-declarations class-member-declaration

```

```

9      class-member-declaration:
10         constant-declaration
11         field-declaration
12         method-declaration
13         property-declaration
14         event-declaration
15         indexer-declaration
16         operator-declaration
17         constructor-declaration
18         destructor-declaration
19         static-constructor-declaration
20         type-declaration

```

21 The members of a class are divided into the following categories:

- 22 • Constants, which represent constant values associated with that class (§17.3).
- 23 • Fields, which are the variables of that class (§17.4).
- 24 • Methods, which implement the computations and actions that can be performed by that class (§17.5).
- 25 • Properties, which define named characteristics and the actions associated with reading and writing those
26 characteristics (§17.6).
- 27 • Events, which define notifications that can be generated by that class (§17.7).
- 28 • Indexers, which permit instances of that class to be indexed in the same way as arrays (§17.8).
- 29 • Operators, which define the expression operators that can be applied to instances of that class (§17.9).
- 30 • Instance constructors, which implement the actions required to initialize instances of that class (§17.10)
- 31 • Destructors, which implement the actions to be performed before instances of that class are permanently
32 discarded (§17.12).
- 33 • Static constructors, which implement the actions required to initialize that class itself (§17.11).
- 34 • Types, which represent the types that are local to that class (§16.5).

35 Members that can contain executable code are collectively known as the *function members* of the class. The
36 function members of a class are the methods, properties, events, indexers, operators, instance constructors,
37 destructors, and static constructors of that class.

38 A *class-declaration* creates a new declaration space (§10.3), and the *class-member-declarations*
39 immediately contained by the *class-declaration* introduce new members into this declaration space. The
40 following rules apply to *class-member-declarations*:

- 41 • Instance constructors, destructors, and static constructors must have the same name as the immediately
42 enclosing class. All other members must have names that differ from the name of the immediately
43 enclosing class.

- 1 • The name of a constant, field, property, event, or type must differ from the names of all other members
2 declared in the same class.
- 3 • The name of a method must differ from the names of all other non-methods declared in the same class.
4 In addition, the signature (§10.6) of a method must differ from the signatures of all other methods
5 declared in the same class.
- 6 • The signature of an instance constructor must differ from the signatures of all other instance constructors
7 declared in the same class.
- 8 • The signature of an indexer must differ from the signatures of all other indexers declared in the same
9 class.
- 10 • The signature of an operator must differ from the signatures of all other operators declared in the same
11 class.

12 The inherited members of a class (§17.2.1) are not part of the declaration space of a class. [*Note:* Thus, a
13 derived class is allowed to declare a member with the same name or signature as an inherited member
14 (which in effect hides the inherited member). *end note*]

15 17.2.1 Inheritance

16 A class *inherits* the members of its direct base class. Inheritance means that a class implicitly contains all
17 members of its direct base class, except for the instance constructors, destructors, and static constructors of
18 the base class. Some important aspects of inheritance are:

- 19 • Inheritance is transitive. If C is derived from B, and B is derived from A, then C inherits the members
20 declared in B as well as the members declared in A.
- 21 • A derived class *extends* its direct base class. A derived class can add new members to those it inherits,
22 but it cannot remove the definition of an inherited member.
- 23 • Instance constructors, destructors, and static constructors are not inherited, but all other members are,
24 regardless of their declared accessibility (§10.5). However, depending on their declared accessibility,
25 inherited members might not be accessible in a derived class.
- 26 • A derived class can *hide* (§10.7.1.2) inherited members by declaring new members with the same name
27 or signature. Note however that hiding an inherited member does not remove that member—it merely
28 makes that member inaccessible in the derived class.
- 29 • An instance of a class contains a set of all instance fields declared in the class and its base classes, and
30 an implicit conversion (§13.1.4) exists from a derived class type to any of its base class types. Thus, a
31 reference to an instance of some derived class can be treated as a reference to an instance of any of its
32 base classes.
- 33 • A class can declare virtual methods, properties, and indexers, and derived classes can override the
34 implementation of these function members. This enables classes to exhibit polymorphic behavior
35 wherein the actions performed by a function member invocation varies depending on the run-time type
36 of the instance through which that function member is invoked.

37 17.2.2 The new modifier

38 A *class-member-declaration* is permitted to declare a member with the same name or signature as an
39 inherited member. When this occurs, the derived class member is said to *hide* the base class member. Hiding
40 an inherited member is not considered an error, but it does cause the compiler to issue a warning. To
41 suppress the warning, the declaration of the derived class member can include a **new** modifier to indicate
42 that the derived member is intended to hide the base member. This topic is discussed further in §10.7.1.2.

43 If a **new** modifier is included in a declaration that doesn't hide an inherited member, a warning to that effect
44 is issued. This warning is suppressed by removing the **new** modifier.

1 17.2.3 Access modifiers

2 A *class-member-declaration* can have any one of the five possible kinds of declared accessibility (§10.5.1):
 3 `public`, `protected internal`, `protected`, `internal`, or `private`. Except for the `protected`
 4 `internal` combination, it is a compile-time error to specify more than one access modifier. When a *class-*
 5 *member-declaration* does not include any access modifiers, `private` is assumed.

6 17.2.4 Constituent types

7 Types that are used in the declaration of a member are called the *constituent types* of that member. Possible
 8 constituent types are the type of a constant, field, property, event, or indexer, the return type of a method or
 9 operator, and the parameter types of a method, indexer, operator, or instance constructor. The constituent
 10 types of a member must be at least as accessible as that member itself (§10.5.4).

11 17.2.5 Static and instance members

12 Members of a class are either *static members* or *instance members*. [*Note*: Generally speaking, it is useful
 13 to think of static members as belonging to classes and instance members as belonging to objects (instances
 14 of classes). *end note*]

15 When a field, method, property, event, operator, or constructor declaration includes a `static` modifier, it
 16 declares a static member. In addition, a constant or type declaration implicitly declares a static member.
 17 Static members have the following characteristics:

- 18 • When a static member is referenced in a *member-access* (§14.5.4) of the form `E.M`, `E` must denote a type
 19 that has a member `M`. It is a compile-time error for `E` to denote an instance.
- 20 • A static field identifies exactly one storage location. No matter how many instances of a class are
 21 created, there is only ever one copy of a static field.
- 22 • A static function member (method, property, event, operator, or constructor) does not operate on a
 23 specific instance, and it is a compile-time error to refer to `this` in such a function member.

24 When a field, method, property, event, indexer, constructor, or destructor declaration does not include a
 25 `static` modifier, it declares an instance member. (An instance member is sometimes called a non-static
 26 member.) Instance members have the following characteristics:

- 27 • When an instance member is referenced in a *member-access* (§14.5.4) of the form `E.M`, `E` must denote
 28 an instance of a type that has a member `M`. It is a compile-time error for `E` to denote a type.
- 29 • Every instance of a class contains a separate set of all instance fields of the class.
- 30 • An instance function member (method, property, indexer, instance constructor, or destructor) operates
 31 on a given instance of the class, and `this` instance can be accessed as `this` (§14.5.7).

32 [*Example*: The following example illustrates the rules for accessing static and instance members:

```

33     class Test
34     {
35         int x;
36         static int y;
37
38         void F() {
39             x = 1;           // Ok, same as this.x = 1
40             y = 1;           // Ok, same as Test.y = 1
41         }
42
43         static void G() {
44             x = 1;           // Error, cannot access this.x
45             y = 1;           // Ok, same as Test.y = 1
46         }
47     }
  
```

```

1         static void Main() {
2             Test t = new Test();
3             t.x = 1;           // Ok
4             t.y = 1;           // Error, cannot access static member through
5 instance
6             Test.x = 1;       // Error, cannot access instance member through type
7             Test.y = 1;       // Ok
8         }
9     }

```

10 The F method shows that in an instance function member, a *simple-name* (§14.5.2) can be used to access
 11 both instance members and static members. The G method shows that in a static function member, it is a
 12 compile-time error to access an instance member through a *simple-name*. The Main method shows that in a
 13 *member-access* (§14.5.4), instance members must be accessed through instances, and static members must
 14 be accessed through types. *end example*]

15 17.2.6 Nested types

16 A type declared within a class or struct is called a *nested type*. A type that is declared within a compilation
 17 unit or namespace is called a *non-nested type*. [*Example*: In the following example:

```

18     using System;
19     class A
20     {
21         class B
22         {
23             static void F() {
24                 Console.WriteLine("A.B.F");
25             }
26         }
27     }

```

28 class B is a nested type because it is declared within class A, and class A is a non-nested type because it is
 29 declared within a compilation unit. *end example*]

30 17.2.6.1 Fully qualified name

31 The fully qualified name (§10.8.1) for a nested type is S.N where S is the fully qualified name of the type in
 32 which type N is declared.

33 17.2.6.2 Declared accessibility

34 Non-nested types can have public or internal declared accessibility and they internal declared accessibility
 35 by default. Nested types can have these forms of declared accessibility too, plus one or more additional
 36 forms of declared accessibility, depending on whether the containing type is a class or struct:

- 37 • A nested type that is declared in a class can have any of five forms of declared accessibility (public,
 38 protected internal, protected, internal, or private) and, like other class members, defaults to private
 39 declared accessibility.
- 40 • A nested type that is declared in a struct can have any of three forms of declared accessibility (public,
 41 internal, or private) and, like other struct members, defaults to private declared accessibility.

42 [*Example*: The example

C# LANGUAGE SPECIFICATION

```
1      public class List
2      {
3          // Private data structure
4          private class Node
5          {
6              public object Data;
7              public Node Next;
8              public Node(object data, Node next) {
9                  this.Data = data;
10                 this.Next = next;
11             }
12         }
13
14         private Node first = null;
15         private Node last = null;
16
17         // Public interface
18         public void AddToFront(object o) {...}
19         public void AddToBack(object o) {...}
20         public object RemoveFromFront() {...}
21         public object AddToFront() {...}
22         public int Count { get {...} }
23     }
```

22 declares a private nested class `Node`. *end example*]

23 17.2.6.3 Hiding

24 A nested type may hide (§10.7.1.1) a base member. The `new` modifier is permitted on nested type
25 declarations so that hiding can be expressed explicitly. [*Example*: The example

```
26     using System;
27     class Base
28     {
29         public static void M() {
30             Console.WriteLine("Base.M");
31         }
32     }
33
34     class Derived: Base
35     {
36         new public class M
37         {
38             public static void F() {
39                 Console.WriteLine("Derived.M.F");
40             }
41         }
42     }
43
44     class Test
45     {
46         static void Main() {
47             Derived.M.F();
48         }
49     }
```

48 shows a nested class `M` that hides the method `M` defined in `Base`. *end example*]

49 17.2.6.4 `this` access

50 A nested type and its containing type do not have a special relationship with regard to *this-access* (§14.5.7).
51 Specifically, `this` within a nested type cannot be used to refer to instance members of the containing type.
52 In cases where a nested type needs access to the instance members of its containing type, access can be
53 provided by providing the `this` for the instance of the containing type as a constructor argument for the
54 nested type. [*Example*: The following example

```

1      using System;
2      class C
3      {
4          int i = 123;
5          public void F() {
6              Nested n = new Nested(this);
7              n.G();
8          }
9
10         public class Nested {
11             C this_c;
12             public Nested(C c) {
13                 this_c = c;
14             }
15             public void G() {
16                 Console.WriteLine(this_c.i);
17             }
18         }
19     }
20     class Test {
21         static void Main() {
22             C c = new C();
23             c.F();
24         }

```

25 shows this technique. An instance of `C` creates an instance of `Nested`, and passes its own `this` to `Nested`'s constructor in order to provide subsequent access to `C`'s instance members. *end example*

27 17.2.6.5 Access to private and protected members of the containing type

28 A nested type has access to all of the members that are accessible to its containing type, including members
29 of the containing type that have private and protected declared accessibility. [*Example*: The example

```

30     using System;
31     class C
32     {
33         private static void F() {
34             Console.WriteLine("C.F");
35         }
36         public class Nested
37         {
38             public static void G() {
39                 F();
40             }
41         }
42     }
43     class Test
44     {
45         static void Main() {
46             C.Nested.G();
47         }
48     }

```

49 shows a class `C` that contains a nested class `Nested`. Within `Nested`, the method `G` calls the static method `F`
50 defined in `C`, and `F` has private declared accessibility. *end example*

51 A nested type also may access protected members defined in a base type of its containing type. [*Example*: In
52 the example

```

53     using System;
54     class Base
55     {
56         protected void F() {
57             Console.WriteLine("Base.F");
58         }
59     }

```

```

1      class Derived: Base
2      {
3          public class Nested
4          {
5              public void G() {
6                  Derived d = new Derived();
7                  d.F();      // ok
8              }
9          }
10     }
11     class Test
12     {
13         static void Main() {
14             Derived.Nested n = new Derived.Nested();
15             n.G();
16         }
17     }

```

18 the nested class `Derived.Nested` accesses the protected method `F` defined in `Derived`'s base class, `Base`,
19 by calling through an instance of `Derived`. *end example*]

20 17.2.7 Reserved member names

21 To facilitate the underlying C# runtime implementation, for each source member declaration that is a
22 property, event, or indexer, the implementation must reserve two method signatures based on the kind of the
23 member declaration, its name, and its type (§17.2.7.1, §17.2.7.2, §17.2.7.3). It is a compile-time error for a
24 program to declare a member whose signature matches one of these reserved signatures, even if the
25 underlying runtime implementation does not make use of these reservations.

26 The reserved names do not introduce declarations, thus they do not participate in member lookup. However,
27 a declaration's associated reserved method signatures do participate in inheritance (§17.2.1), and can be
28 hidden with the `new` modifier (§17.2.2).

29 [*Note:* The reservation of these names serves three purposes:

- 30 1. To allow the underlying implementation to use an ordinary identifier as a method name for get or set
31 access to the C# language feature.
- 32 2. To allow other languages to interoperate using an ordinary identifier as a method name for get or set
33 access to the C# language feature.
- 34 3. To help ensure that the source accepted by one conforming compiler is accepted by another, by
35 making the specifics of reserved member names consistent across all C# implementations.

36 *end note*]

37 The declaration of a destructor (§17.12) also causes a signature to be reserved (§17.2.7.4).

38 17.2.7.1 Member Names Reserved for Properties

39 For a property `P` (§17.6) of type `T`, the following signatures are reserved:

```

40     T get_P();
41     void set_P(T value);

```

42 Both signatures are reserved, even if the property is read-only or write-only.

43 [*Example:* In the example

```

44     using System;
45     class A {
46         public int P {
47             get { return 123; }
48         }
49     }

```



```

1      class B: A {
2          new public int get_P() {
3              return 456;
4          }
5          new public void set_P(int value) {
6          }
7      }

8      class Test
9      {
10         static void Main() {
11             B b = new B();
12             A a = b;
13             Console.WriteLine(a.P);
14             Console.WriteLine(b.P);
15             Console.WriteLine(b.get_P());
16         }
17     }

```

18 a class A defines a read-only property P, thus reserving signatures for `get_P` and `set_P` methods. A class B
19 derives from A and hides both of these reserved signatures. The example produces the output:

```

20         123
21         123
22         456

```

23 *end example]*

24 17.2.7.2 Member Names Reserved for Events

25 For an event E (§17.7) of delegate type T, the following signatures are reserved:

```

26         void add_E(T handler);
27         void remove_E(T handler);

```

28 17.2.7.3 Member Names Reserved for Indexers

29 For an indexer (§17.8) of type T with parameter-list L, the following signatures are reserved:

```

30         T get_Item(L);
31         void set_Item(L, T value);

```

32 Both signatures are reserved, even if the indexer is read-only or write-only.

33 17.2.7.4 Member Names Reserved for Destructors

34 For a class containing a destructor (§17.12), the following signature is reserved:

```

35         void Finalize();

```

36 17.3 Constants

37 A **constant** is a class member that represents a constant value: a value that can be computed at compile-time.

38 A *constant-declaration* introduces one or more constants of a given type.

39 *constant-declaration:*

```

40         attributesopt constant-modifiersopt const type constant-declarators ;

```

41 *constant-modifiers:*

```

42         constant-modifier

```

```

43         constant-modifiers constant-modifier

```

44 *constant-modifier:*

```

45         new

```

```

46         public

```

```

47         protected

```

```

48         internal

```

```

49         private

```

C# LANGUAGE SPECIFICATION

```
1      constant-declarators:  
2          constant-declarator  
3          constant-declarators , constant-declarator
```

```
4      constant-declarator:  
5          identifier = constant-expression
```

6 A *constant-declaration* may include a set of *attributes* (§24), a *new* modifier (§17.2.2), and a valid
7 combination of the four access modifiers (§17.2.3). The attributes and modifiers apply to all of the members
8 declared by the *constant-declaration*. Even though constants are considered static members, a *constant-*
9 *declaration* neither requires nor allows a `static` modifier. It is an error for the same modifier to appear
10 multiple times in a constant declaration.

11 The *type* of a *constant-declaration* specifies the type of the members introduced by the declaration. The type
12 is followed by a list of *constant-declarators*, each of which introduces a new member. A *constant-declarator*
13 consists of an *identifier* that names the member, followed by an “=” token, followed by a *constant-*
14 *expression* (§14.15) that gives the value of the member.

15 The *type* specified in a constant declaration must be `sbyte`, `byte`, `short`, `ushort`, `int`, `uint`, `long`,
16 `ulong`, `char`, `float`, `double`, `decimal`, `bool`, `string`, an *enum-type*, or a *reference-type*. Each *constant-*
17 *expression* must yield a value of the target type or of a type that can be converted to the target type by an
18 implicit conversion (§13.1).

19 The *type* of a constant must be at least as accessible as the constant itself (§10.5.4).

20 The value of a constant is obtained in an expression using a *simple-name* (§14.5.2) or a *member-access*
21 (§14.5.4).

22 A constant can itself participate in a *constant-expression*. Thus, a constant may be used in any construct that
23 requires a *constant-expression*. [Note: Examples of such constructs include `case` labels, `goto case`
24 statements, `enum` member declarations, attributes, and other constant declarations. *end note*]

25 [Note: As described in §14.15, a *constant-expression* is an expression that can be fully evaluated at compile-
26 time. Since the only way to create a non-null value of a *reference-type* other than `string` is to apply the
27 `new` operator, and since the `new` operator is not permitted in a *constant-expression*, the only possible value
28 for constants of *reference-types* other than `string` is `null`. *end note*]

29 When a symbolic name for a constant value is desired, but when the type of that value is not permitted in a
30 constant declaration, or when the value cannot be computed at compile-time by a *constant-expression*, a
31 `readonly` field (§17.4.2) may be used instead. [Note: The versioning semantics of `const` and `readonly`
32 differ (§17.4.2.2). *end-note*]

33 A constant declaration that declares multiple constants is equivalent to multiple declarations of single
34 constants with the same attributes, modifiers, and type. [Example: For example

```
35     class A  
36     {  
37         public const double x = 1.0, y = 2.0, z = 3.0;  
38     }
```

39 is equivalent to

```
40     class A  
41     {  
42         public const double x = 1.0;  
43         public const double y = 2.0;  
44         public const double z = 3.0;  
45     }
```

46 *end example*]

47 Constants are permitted to depend on other constants within the same program as long as the dependencies
48 are not of a circular nature. The compiler automatically arranges to evaluate the constant declarations in the
49 appropriate order. [Example: In the example

```

1      class A
2      {
3          public const int X = B.Z + 1;
4          public const int Y = 10;
5      }
6
7      class B
8      {
9          public const int Z = A.Y + 1;
10     }

```

10 the compiler first evaluates `A.Y`, then evaluates `B.Z`, and finally evaluates `A.X`, producing the values 10, 11, and 12. *end example*] Constant declarations may depend on constants from other programs, but such dependencies are only possible in one direction. [*Example:* Referring to the example above, if A and B were declared in separate programs, it would be possible for `A.X` to depend on `B.Z`, but `B.Z` could then not simultaneously depend on `A.Y`. *end example*]

15 17.4 Fields

16 A *field* is a member that represents a variable associated with an object or class. A *field-declaration* introduces one or more fields of a given type.

```

18     field-declaration:
19         attributesopt field-modifiersopt type variable-declarators ;
20
21     field-modifiers:
22         field-modifier
23         field-modifiers field-modifier
24
25     field-modifier:
26         new
27         public
28         protected
29         internal
30         private
31         static
32         readonly
33         volatile
34
35     variable-declarators:
36         variable-declarator
37         variable-declarators , variable-declarator
38
39     variable-declarator:
40         identifier
41         identifier = variable-initializer
42
43     variable-initializer:
44         expression
45         array-initializer

```

41 A *field-declaration* may include a set of *attributes* (§24), a `new` modifier (§17.2.2), a valid combination of the four access modifiers (§17.2.3), and a `static` modifier (§17.4.1). In addition, a *field-declaration* may include a `readonly` modifier (§17.4.2) or a `volatile` modifier (§17.4.3), but not both. The attributes and modifiers apply to all of the members declared by the *field-declaration*. It is an error for the same modifier to appear multiple times in a *field declaration*. It is an error for the same modifier to appear multiple times in a field declaration.

47 The *type* of a *field-declaration* specifies the type of the members introduced by the declaration. The type is followed by a list of *variable-declarators*, each of which introduces a new member. A *variable-declarator* consists of an *identifier* that names that member, optionally followed by an “=” token and a *variable-initializer* (§17.4.5) that gives the initial value of that member.

1 The *type* of a field must be at least as accessible as the field itself (§10.5.4).

2 The value of a field is obtained in an expression using a *simple-name* (§14.5.2) or a *member-access*
 3 (§14.5.4). The value of a non-readonly field is modified using an *assignment* (§14.13). The value of a non-
 4 readonly field can be both obtained and modified using postfix increment and decrement operators (§14.5.9)
 5 and prefix increment and decrement operators (§14.6.5).

6 A field declaration that declares multiple fields is equivalent to multiple declarations of single fields with the
 7 same attributes, modifiers, and type. [*Example*: For example

```
8     class A
9     {
10        public static int x = 1, Y, Z = 100;
11    }
```

12 is equivalent to

```
13     class A
14     {
15        public static int x = 1;
16        public static int Y;
17        public static int Z = 100;
18    }
```

19 *end example*]

20 **17.4.1 Static and instance fields**

21 When a field declaration includes a `static` modifier, the fields introduced by the declaration are *static*
 22 *fields*. When no `static` modifier is present, the fields introduced by the declaration are *instance fields*.
 23 Static fields and instance fields are two of the several kinds of variables (§12) supported by C#, and at times
 24 they are referred to as *static variables* and *instance variables*, respectively.

25 A static field is not part of a specific instance; instead, it identifies exactly one storage location. No matter
 26 how many instances of a class are created, there is only ever one copy of a static field for the associated
 27 application domain.

28 An instance field belongs to an instance. Specifically, every instance of a class contains a separate set of all
 29 the instance fields of that class.

30 When a field is referenced in a *member-access* (§14.5.4) of the form `E.M`, if `M` is a static field, `E` must denote
 31 a type that has a field `M`, and if `M` is an instance field, `E` must denote an instance of a type that has a field `M`.

32 The differences between static and instance members are discussed further in §17.2.5.

33 **17.4.2 Readonly fields**

34 When a *field-declaration* includes a `readonly` modifier, the fields introduced by the declaration are
 35 *readonly fields*. Direct assignments to readonly fields can only occur as part of that declaration or in an
 36 instance constructor or static constructor in the same class. (A readonly field can be assigned to multiple
 37 times in these contexts.) Specifically, direct assignments to a `readonly` field are permitted only in the
 38 following contexts:

- 39 • In the *variable-declarator* that introduces the field (by including a *variable-initializer* in the
 40 declaration).
- 41 • For an instance field, in the instance constructors of the class that contains the field declaration; for a
 42 static field, in the static constructor of the class that contains the field declaration. These are also the
 43 only contexts in which it is valid to pass a `readonly` field as an `out` or `ref` parameter.

44 Attempting to assign to a `readonly` field or pass it as an `out` or `ref` parameter in any other context is a
 45 compile-time error.

1 17.4.2.1 Using static readonly fields for constants

2 A `static readonly` field is useful when a symbolic name for a constant value is desired, but when the
3 type of the value is not permitted in a `const` declaration, or when the value cannot be computed at compile-
4 time. [*Example*: In the example

```
5     public class Color
6     {
7         public static readonly Color Black = new Color(0, 0, 0);
8         public static readonly Color White = new Color(255, 255, 255);
9         public static readonly Color Red = new Color(255, 0, 0);
10        public static readonly Color Green = new Color(0, 255, 0);
11        public static readonly Color Blue = new Color(0, 0, 255);
12
13        private byte red, green, blue;
14
15        public Color(byte r, byte g, byte b) {
16            red = r;
17            green = g;
18            blue = b;
19        }
20    }
```

19 the `Black`, `White`, `Red`, `Green`, and `Blue` members cannot be declared as `const` members because their
20 values cannot be computed at compile-time. However, declaring them `static readonly` instead has much
21 the same effect. [*end example*]

22 17.4.2.2 Versioning of constants and static readonly fields

23 Constants and readonly fields have different binary versioning semantics. When an expression references a
24 constant, the value of the constant is obtained at compile-time, but when an expression references a readonly
25 field, the value of the field is not obtained until run-time. [*Example*: Consider an application that consists of
26 two separate programs:

```
27     using System;
28     namespace Program1
29     {
30         public class Utils
31         {
32             public static readonly int X = 1;
33         }
34     }
35
36     namespace Program2
37     {
38         class Test
39         {
40             static void Main() {
41                 Console.WriteLine(Program1.Utils.X);
42             }
43         }
44     }
```

44 The `Program1` and `Program2` namespaces denote two programs that are compiled separately. Because
45 `Program1.Utils.X` is declared as a static readonly field, the value output by the `Console.WriteLine`
46 statement is not known at compile-time, but rather is obtained at run-time. Thus, if the value of `X` is changed
47 and `Program1` is recompiled, the `Console.WriteLine` statement will output the new value even if
48 `Program2` isn't recompiled. However, had `X` been a constant, the value of `X` would have been obtained at
49 the time `Program2` was compiled, and would remain unaffected by changes in `Program1` until `Program2`
50 is recompiled. [*end example*]

51 17.4.3 Volatile fields

52 When a *field-declaration* includes a `volatile` modifier, the fields introduced by that declaration are
53 *volatile fields*. For non-volatile fields, optimization techniques that reorder instructions can lead to
54 unexpected and unpredictable results in multi-threaded programs that access fields without synchronization

1 such as that provided by the *lock-statement* (§15.12). These optimizations can be performed by the compiler,
2 by the runtime system, or by hardware. For volatile fields, such reordering optimizations are restricted:

- 3 • A read of a volatile field is called a *volatile read*. A volatile read has “acquire semantics”; that is, it is
4 guaranteed to occur prior to any references to memory that occur after it in the instruction sequence.
- 5 • A write of a volatile field is called a *volatile write*. A volatile write has “release semantics”; that is, it is
6 guaranteed to happen after any memory references prior to the write instruction in the instruction
7 sequence.

8 These restrictions ensure that all threads will observe volatile writes performed by any other thread in the
9 order in which they were performed. A conforming implementation is not required to provide a single total
10 ordering of volatile writes as seen from all threads of execution. The type of a volatile field must be one of
11 the following:

- 12 • A *reference-type*.
- 13 • The type `byte`, `sbyte`, `short`, `ushort`, `int`, `uint`, `char`, `float`, or `bool`.
- 14 • An *enum-type* having an enum base type of `byte`, `sbyte`, `short`, `ushort`, `int`, or `uint`.

15 [*Example:* The example

```
16     using System;
17     using System.Threading;
18     class Test
19     {
20         public static int result;
21         public static volatile bool finished;
22         static void Thread2() {
23             result = 143;
24             finished = true;
25         }
26         static void Main() {
27             finished = false;
28             // Run Thread2() in a new thread
29             new Thread(new ThreadStart(Thread2)).Start();
30             // wait for Thread2 to signal that it has a result by setting
31             // finished to true.
32             for (;;) {
33                 if (finished) {
34                     Console.WriteLine("result = {0}", result);
35                     return;
36                 }
37             }
38         }
39     }
```

40 produces the output:

```
41     result = 143
```

42 In this example, the method `Main` starts a new thread that runs the method `Thread2`. This method stores a
43 value into a non-volatile field called `result`, then stores `true` in the volatile field `finished`. The main
44 thread waits for the field `finished` to be set to `true`, then reads the field `result`. Since `result` has been
45 declared `volatile`, the main thread must read the value 143 from the field `result`. If the field `finished`
46 had not been declared `volatile`, then it would be permissible for the store to `result` to be visible to the
47 main thread *after* the store to `finished`, and hence for the main thread to read the value 0 from the field
48 `result`. Declaring `finished` as a `volatile` field prevents any such inconsistency. *end example*]

49 17.4.4 Field initialization

50 The initial value of a field, whether it be a static field or an instance field, is the default value (§12.2) of the
51 field’s type. It is not possible to observe the value of a field before this default initialization has occurred,
52 and a field is thus never “uninitialized”. [*Example:* The example

```

1      using System;
2      class Test
3      {
4          static bool b;
5          int i;
6
7          static void Main() {
8              Test t = new Test();
9              Console.WriteLine("b = {0}, i = {1}", b, t.i);
10         }
11     }

```

11 produces the output

```
12     b = False, i = 0
```

13 because `b` and `i` are both automatically initialized to default values. *end example*]

14 17.4.5 Variable initializers

15 Field declarations may include *variable-initializers*. For static fields, variable initializers correspond to
16 assignment statements that are executed during class initialization. For instance fields, variable initializers
17 correspond to assignment statements that are executed when an instance of the class is created.

18 [*Example:* The example

```

19     using System;
20     class Test
21     {
22         static double x = Math.Sqrt(2.0);
23         int i = 100;
24         string s = "Hello";
25
26         static void Main() {
27             Test a = new Test();
28             Console.WriteLine("x = {0}, i = {1}, s = {2}", x, a.i, a.s);
29         }
30     }

```

30 produces the output

```
31     x = 1.4142135623731, i = 100, s = Hello
```

32 because an assignment to `x` occurs when static field initializers execute and assignments to `i` and `s` occur
33 when the instance field initializers execute. *end example*]

34 The default value initialization described in §17.4.3 occurs for all fields, including fields that have variable
35 initializers. Thus, when a class is initialized, all static fields in that class are first initialized to their default
36 values, and then the static field initializers are executed in textual order. Likewise, when an instance of a
37 class is created, all instance fields in that instance are first initialized to their default values, and then the
38 instance field initializers are executed in textual order.

39 It is possible for static fields with variable initializers to be observed in their default value state. [*Example:*
40 However, this is strongly discouraged as a matter of style. The example

```

41     using System;
42     class Test
43     {
44         static int a = b + 1;
45         static int b = a + 1;
46
47         static void Main() {
48             Console.WriteLine("a = {0}, b = {1}", a, b);
49         }
50     }

```

50 exhibits this behavior. Despite the circular definitions of `a` and `b`, the program is valid. It results in the output

```
51     a = 1, b = 2
```

1 because the static fields `a` and `b` are initialized to 0 (the default value for `int`) before their initializers are
 2 executed. When the initializer for `a` runs, the value of `b` is zero, and so `a` is initialized to 1. When the
 3 initializer for `b` runs, the value of `a` is already 1, and so `b` is initialized to 2. *end example*]

4 17.4.5.1 Static field initialization

5 The static field variable initializers of a class correspond to a sequence of assignments that are executed in
 6 the textual order in which they appear in the class declaration. If a static constructor (§17.11) exists in the
 7 class, execution of the static field initializers occurs immediately prior to executing that static constructor.
 8 Otherwise, the static field initializers are executed at an implementation-dependent time prior to the first use
 9 of a static field of that class. [*Example*: The example

```
10     using System;
11     class Test
12     {
13         static void Main() {
14             Console.WriteLine("{0} {1}", B.Y, A.X);
15         }
16         public static int f(string s) {
17             Console.WriteLine(s);
18             return 1;
19         }
20     }
21     class A
22     {
23         public static int X = Test.f("Init A");
24     }
25     class B
26     {
27         public static int Y = Test.f("Init B");
28     }
```

29 might produce either the output:

```
30     Init A
31     Init B
32     1 1
```

33 or the output:

```
34     Init B
35     Init A
36     1 1
```

37 because the execution of `X`'s initializer and `Y`'s initializer could occur in either order; they are only
 38 constrained to occur before the references to those fields. However, in the example:

```
39     using System;
40     class Test {
41         static void Main() {
42             Console.WriteLine("{0} {1}", B.Y, A.X);
43         }
44         public static int f(string s) {
45             Console.WriteLine(s);
46             return 1;
47         }
48     }
49     class A
50     {
51         static A() {}
52         public static int X = Test.f("Init A");
53     }
```



```

1      class B
2      {
3          static B() {}
4          public static int Y = Test.f("Init B");
5      }

```

6 the output must be:

```

7      Init B
8      Init A
9      1 1

```

10 because the rules for when static constructors execute provide that B's static constructor (and hence B's static
 11 field initializers) must run before A's static constructor and field initializers. *end example*]

12 17.4.5.2 Instance field initialization

13 The instance field variable initializers of a class correspond to a sequence of assignments that are executed
 14 immediately upon entry to any one of the instance constructors (§17.10.2) of that class. The variable
 15 initializers are executed in the textual order in which they appear in the class declaration. The class instance
 16 creation and initialization process is described further in §17.10.

17 A variable initializer for an instance field cannot reference the instance being created. Thus, it is a compile-
 18 time error to reference `this` in a variable initializer, as it is a compile-time error for a variable initializer to
 19 reference any instance member through a *simple-name*. [*Example*: In the example

```

20      class A
21      {
22          int x = 1;
23          int y = x + 1;    // Error, reference to instance member of this
24      }

```

25 the variable initializer for `y` results in a compile-time error because it references a member of the instance
 26 being created. *end example*]

27 17.5 Methods

28 A *method* is a member that implements a computation or action that can be performed by an object or class.
 29 Methods are declared using *method-declarations*:

30 *method-declaration*:

31 *method-header method-body*

32 *method-header*:

33 *attributes*_{opt} *method-modifiers*_{opt} *return-type* *member-name* (*formal-parameter-list*_{opt})

34 *method-modifiers*:

35 *method-modifier*

36 *method-modifiers method-modifier*

37 *method-modifier*:

38 `new`

39 `public`

40 `protected`

41 `internal`

42 `private`

43 `static`

44 `virtual`

45 `sealed`

46 `override`

47 `abstract`

48 `extern`

```

1      return-type:
2          type
3          void
4
5      member-name:
6          identifier
7          interface-type . identifier
8
9      method-body:
10         block
11         ;

```

10 A *method-declaration* may include a set of *attributes* (§24) and a valid combination of the four access
11 modifiers (§17.2.3), the **new** (§17.2.2), **static** (§17.5.2), **virtual** (§17.5.3), **override** (§17.5.4),
12 **sealed** (§17.5.5), **abstract** (§17.5.6), and **extern** (§17.5.7) modifiers.

13 A declaration has a valid combination of modifiers if all of the following are true:

- 14 • The declaration includes a valid combination of access modifiers (§17.2.3).
- 15 • The declaration does not include the same modifier multiple times.
- 16 • The declaration includes at most one of the following modifiers: **static**, **virtual**, and **override**.
- 17 • The declaration includes at most one of the following modifiers: **new** and **override**.
- 18 • If the declaration includes the **abstract** modifier, then the declaration does not include any of the
19 following modifiers: **static**, **virtual**, or **extern**.
- 20 • If the declaration includes the **private** modifier, then the declaration does not include any of the
21 following modifiers: **virtual**, **override**, or **abstract**.
- 22 • If the declaration includes the **sealed** modifier, then the declaration also includes the **override**
23 modifier.

24 The *return-type* of a method declaration specifies the type of the value computed and returned by the
25 method. The *return-type* is **void** if the method does not return a value.

26 The *member-name* specifies the name of the method. Unless the method is an explicit interface member
27 implementation (§20.4.1), the *member-name* is simply an *identifier*. For an explicit interface member
28 implementation, the *member-name* consists of an *interface-type* followed by a “.” and an *identifier*.

29 The optional *formal-parameter-list* specifies the parameters of the method (§17.5.1).

30 The *return-type* and each of the types referenced in the *formal-parameter-list* of a method must be at least as
31 accessible as the method itself (§10.5.4).

32 For **abstract** and **extern** methods, the *method-body* consists simply of a semicolon. For all other
33 methods, the *method-body* consists of a *block*, which specifies the statements to execute when the method is
34 invoked.

35 The name and the formal parameter list of a method define the signature (§10.6) of the method. Specifically,
36 the signature of a method consists of its name and the number, modifiers, and types of its formal parameters.
37 The return type is not part of a method’s signature, nor are the names of the formal parameters.

38 The name of a method must differ from the names of all other non-methods declared in the same class. In
39 addition, the signature of a method must differ from the signatures of all other methods declared in the same
40 class.

41 **17.5.1 Method parameters**

42 The parameters of a method, if any, are declared by the method’s *formal-parameter-list*.

```

1      formal-parameter-list:
2          fixed-parameters
3          fixed-parameters , parameter-array
4          parameter-array
5
6      fixed-parameters:
7          fixed-parameter
8          fixed-parameters , fixed-parameter
9
10     fixed-parameter:
11         attributesopt parameter-modifieropt type identifier
12
13     parameter-modifier:
14         ref
15         out
16
17     parameter-array:
18         attributesopt params array-type identifier

```

The formal parameter list consists of one or more comma-separated parameters of which only the last may be a *parameter-array*.

A *fixed-parameter* consists of an optional set of *attributes* (§24), an optional **ref** or **out** modifier, a *type*, and an *identifier*. Each *fixed-parameter* declares a parameter of the given type with the given name.

A *parameter-array* consists of an optional set of *attributes* (§24), a **params** modifier, an *array-type*, and an *identifier*. A parameter array declares a single parameter of the given array type with the given name. The *array-type* of a parameter array must be a single-dimensional array type (§19.1). In a method invocation, a parameter array permits either a single argument of the given array type to be specified, or it permits zero or more arguments of the array element type to be specified. Parameter arrays are described further in §17.5.1.4.

A method declaration creates a separate declaration space for parameters and local variables. Names are introduced into this declaration space by the formal parameter list of the method and by local variable declarations in the *block* of the method. All names in the declaration space of a method must be unique. Thus, it is a compile-time error for a parameter or local variable to have the same name as another parameter or local variable.

A method invocation (§14.5.5.1) creates a copy, specific to that invocation, of the formal parameters and local variables of the method, and the argument list of the invocation assigns values or variable references to the newly created formal parameters. Within the *block* of a method, formal parameters can be referenced by their identifiers in *simple-name* expressions (§14.5.2).

There are four kinds of formal parameters:

- Value parameters, which are declared without any modifiers.
- Reference parameters, which are declared with the **ref** modifier.
- Output parameters, which are declared with the **out** modifier.
- Parameter arrays, which are declared with the **params** modifier.

[*Note*: As described in §10.6, the **ref** and **out** modifiers are part of a method's signature, but the **params** modifier is not. *end note*]

17.5.1.1 Value parameters

A parameter declared with no modifiers is a value parameter. A value parameter corresponds to a local variable that gets its initial value from the corresponding argument supplied in the method invocation.

When a formal parameter is a value parameter, the corresponding argument in a method invocation must be an expression of a type that is implicitly convertible (§13.1) to the formal parameter type.

1 A method is permitted to assign new values to a value parameter. Such assignments only affect the local
 2 storage location represented by the value parameter—they have no effect on the actual argument given in the
 3 method invocation.

4 17.5.1.2 Reference parameters

5 A parameter declared with a `ref` modifier is a reference parameter. Unlike a value parameter, a reference
 6 parameter does not create a new storage location. Instead, a reference parameter represents the same storage
 7 location as the variable given as the argument in the method invocation.

8 When a formal parameter is a reference parameter, the corresponding argument in a method invocation must
 9 consist of the keyword `ref` followed by a *variable-reference* (§12.3.3) of the same type as the formal
 10 parameter. A variable must be definitely assigned before it can be passed as a reference parameter.

11 Within a method, a reference parameter is always considered definitely assigned.

12 [*Example:* The example

```

13     using System;
14     class Test
15     {
16         static void Swap(ref int x, ref int y) {
17             int temp = x;
18             x = y;
19             y = temp;
20         }
21
22         static void Main() {
23             int i = 1, j = 2;
24             Swap(ref i, ref j);
25             Console.WriteLine("i = {0}, j = {1}", i, j);
26         }
27     }
  
```

27 produces the output

```

28     i = 2, j = 1
  
```

29 For the invocation of `Swap` in `Main`, `x` represents `i` and `y` represents `j`. Thus, the invocation has the effect of
 30 swapping the values of `i` and `j`. *end example*]

31 In a method that takes reference parameters, it is possible for multiple names to represent the same storage
 32 location. [*Example:* In the example

```

33     class A
34     {
35         string s;
36
37         void F(ref string a, ref string b) {
38             s = "One";
39             a = "Two";
40             b = "Three";
41         }
42
43         void G() {
44             F(ref s, ref s);
45         }
46     }
  
```

45 the invocation of `F` in `G` passes a reference to `s` for both `a` and `b`. Thus, for that invocation, the names `s`, `a`,
 46 and `b` all refer to the same storage location, and the three assignments all modify the instance field `s`. *end*
 47 *example*]

48 17.5.1.3 Output parameters

49 A parameter declared with an `out` modifier is an output parameter. Similar to a reference parameter, an
 50 output parameter does not create a new storage location. Instead, an output parameter represents the same
 51 storage location as the variable given as the argument in the method invocation.

1 When a formal parameter is an output parameter, the corresponding argument in a method invocation must
 2 consist of the keyword `out` followed by a *variable-reference* (§12.3.3) of the same type as the formal
 3 parameter. A variable need not be definitely assigned before it can be passed as an output parameter, but
 4 following an invocation where a variable was passed as an output parameter, the variable is considered
 5 definitely assigned.

6 Within a method, just like a local variable, an output parameter is initially considered unassigned and must
 7 be definitely assigned before its value is used.

8 Every output parameter of a method must be definitely assigned before the method returns.

9 Output parameters are typically used in methods that produce multiple return values. [*Example*: For
 10 example:

```

11     using System;
12     class Test
13     {
14         static void SplitPath(string path, out string dir, out string name) {
15             int i = path.Length;
16             while (i > 0) {
17                 char ch = path[i - 1];
18                 if (ch == '\\\' || ch == '/' || ch == ':') break;
19                 i--;
20             }
21             dir = path.Substring(0, i);
22             name = path.Substring(i);
23         }
24         static void Main() {
25             string dir, name;
26             SplitPath("c:\\windows\\system\\hello.txt", out dir, out name);
27             Console.WriteLine(dir);
28             Console.WriteLine(name);
29         }
30     }
  
```

31 The example produces the output:

```

32     c:\windows\system\
33     hello.txt
  
```

34 Note that the `dir` and `name` variables can be unassigned before they are passed to `SplitPath`, and that they
 35 are considered definitely assigned following the call. *end example*]

36 17.5.1.4 Parameter arrays

37 A parameter declared with a `params` modifier is a parameter array. If a formal parameter list includes a
 38 parameter array, it must be the last parameter in the list and it must be of a single-dimensional array type.
 39 [*Example*: For example, the types `string[]` and `string[][]` can be used as the type of a parameter array,
 40 but the type `string[,]` can not. *end example*] It is not possible to combine the `params` modifier with the
 41 modifiers `ref` and `out`.

42 A parameter array permits arguments to be specified in one of two ways in a method invocation:

- 43 • The argument given for a parameter array can be a single expression of a type that is implicitly
 44 convertible (§13.1) to the parameter array type. In this case, the parameter array acts precisely like a
 45 value parameter.
- 46 • Alternatively, the invocation can specify zero or more arguments for the parameter array, where each
 47 argument is an expression of a type that is implicitly convertible (§13.1) to the element type of the
 48 parameter array. In this case, the invocation creates an instance of the parameter array type with a length
 49 corresponding to the number of arguments, initializes the elements of the array instance with the given
 50 argument values, and uses the newly created array instance as the actual argument.

51 Except for allowing a variable number of arguments in an invocation, a parameter array is precisely
 52 equivalent to a value parameter (§17.5.1.1) of the same type.

C# LANGUAGE SPECIFICATION

1 *[Example:* The example

```
2     using System;
3     class Test
4     {
5         static void F(params int[] args) {
6             Console.WriteLine("Array contains {0} elements:", args.Length);
7             foreach (int i in args)
8                 Console.WriteLine(" {0}", i);
9             Console.WriteLine();
10        }
11
12        static void Main() {
13            int[] arr = {1, 2, 3};
14            F(arr);
15            F(10, 20, 30, 40);
16            F();
17        }
18    }
```

18 produces the output

```
19     Array contains 3 elements: 1 2 3
20     Array contains 4 elements: 10 20 30 40
21     Array contains 0 elements:
```

22 The first invocation of `F` simply passes the array `a` as a value parameter. The second invocation of `F`
23 automatically creates a four-element `int[]` with the given element values and passes that array instance as a
24 value parameter. Likewise, the third invocation of `F` creates a zero-element `int[]` and passes that instance
25 as a value parameter. The second and third invocations are precisely equivalent to writing:

```
26     F(new int[] {10, 20, 30, 40});
27     F(new int[] {});
```

28 *end example]*

29 When performing overload resolution, a method with a parameter array may be applicable either in its
30 normal form or in its expanded form (§14.4.2.1). The expanded form of a method is available only if the
31 normal form of the method is not applicable and only if a method with the same signature as the expanded
32 form is not already declared in the same type.

33 *[Example:* The example

```
34     using System;
35     class Test
36     {
37         static void F(params object[] a) {
38             Console.WriteLine("F(object[])");
39         }
40
41         static void F() {
42             Console.WriteLine("F()");
43         }
44
45         static void F(object a0, object a1) {
46             Console.WriteLine("F(object,object)");
47         }
48
49         static void Main() {
50             F();
51             F(1);
52             F(1, 2);
53             F(1, 2, 3);
54             F(1, 2, 3, 4);
55         }
56     }
```

54 produces the output

```

1      F();
2      F(object[]);
3      F(object,object);
4      F(object[]);
5      F(object[]);

```

6 In the example, two of the possible expanded forms of the method with a parameter array are already
7 included in the class as regular methods. These expanded forms are therefore not considered when
8 performing overload resolution, and the first and third method invocations thus select the regular methods.
9 When a class declares a method with a parameter array, it is not uncommon to also include some of the
10 expanded forms as regular methods. By doing so it is possible to avoid the allocation of an array instance
11 that occurs when an expanded form of a method with a parameter array is invoked. *end example*

12 When the type of a parameter array is `object[]`, a potential ambiguity arises between the normal form of
13 the method and the expanded form for a single `object` parameter. The reason for the ambiguity is that an
14 `object[]` is itself implicitly convertible to type `object`. The ambiguity presents no problem, however,
15 since it can be resolved by inserting a cast if needed.

16 *[Example: The example*

```

17      using System;
18      class Test
19      {
20          static void F(params object[] args) {
21              foreach (object o in args) {
22                  Console.Write(o.GetType().FullName);
23                  Console.Write(" ");
24              }
25              Console.WriteLine();
26          }
27          static void Main() {
28              object[] a = {1, "Hello", 123.456};
29              object o = a;
30              F(a);
31              F((object)a);
32              F(o);
33              F((object[])o);
34          }
35      }

```

36 produces the output

```

37      System.Int32 System.String System.Double
38      System.Object[]
39      System.Object[]
40      System.Int32 System.String System.Double

```

41 In the first and last invocations of `F`, the normal form of `F` is applicable because an implicit conversion exists
42 from the argument type to the parameter type (both are of type `object[]`). Thus, overload resolution selects
43 the normal form of `F`, and the argument is passed as a regular value parameter. In the second and third
44 invocations, the normal form of `F` is not applicable because no implicit conversion exists from the argument
45 type to the parameter type (type `object` cannot be implicitly converted to type `object[]`). However, the
46 expanded form of `F` is applicable, so it is selected by overload resolution. As a result, a one-element
47 `object[]` is created by the invocation, and the single element of the array is initialized with the given
48 argument value (which itself is a reference to an `object[]`). *end example*

49 17.5.2 Static and instance methods

50 When a method declaration includes a `static` modifier, that method is said to be a static method. When no
51 `static` modifier is present, the method is said to be an instance method.

52 A static method does not operate on a specific instance, and it is a compile-time error to refer to `this` in a
53 static method.

1 An instance method operates on a given instance of a class, and that instance can be accessed as `this`
 2 (§14.5.7).

3 When a method is referenced in a *member-access* (§14.5.4) of the form `E.M`, if `M` is a static method, `E` must
 4 denote a type that has a method `M`, and if `M` is an instance method, `E` must denote an instance of a type that
 5 has a method `M`.

6 The differences between static and instance members are discussed further in §17.2.5.

7 **17.5.3 Virtual methods**

8 When an instance method declaration includes a `virtual` modifier, that method is said to be a *virtual*
 9 *method*. When no `virtual` modifier is present, the method is said to be a *non-virtual method*.

10 The implementation of a non-virtual method is invariant: The implementation is the same whether the
 11 method is invoked on an instance of the class in which it is declared or an instance of a derived class. In
 12 contrast, the implementation of a virtual method can be superseded by derived classes. The process of
 13 superseding the implementation of an inherited virtual method is known as *overriding* that method (§17.5.4).

14 In a virtual method invocation, the *run-time type* of the instance for which that invocation takes place
 15 determines the actual method implementation to invoke. In a non-virtual method invocation, the *compile-*
 16 *time type* of the instance is the determining factor. In precise terms, when a method named `N` is invoked with
 17 an argument list `A` on an instance with a compile-time type `C` and a run-time type `R` (where `R` is either `C` or a
 18 class derived from `C`), the invocation is processed as follows:

- 19 • First, overload resolution is applied to `C`, `N`, and `A`, to select a specific method `M` from the set of methods
 20 declared in and inherited by `C`. This is described in §14.5.5.1.
- 21 • Then, if `M` is a non-virtual method, `M` is invoked.
- 22 • Otherwise, `M` is a virtual method, and the most derived implementation of `M` with respect to `R` is invoked.

23 For every virtual method declared in or inherited by a class, there exists a *most derived implementation* of
 24 the method with respect to that class. The most derived implementation of a virtual method `M` with respect to
 25 a class `R` is determined as follows:

- 26 • If `R` contains the introducing `virtual` declaration of `M`, then this is the most derived implementation
 27 of `M`.
- 28 • Otherwise, if `R` contains an `override` of `M`, then this is the most derived implementation of `M`.
- 29 • Otherwise, the most derived implementation of `M` is the same as that of the direct base class of `R`.

30 [*Example:* The following example illustrates the differences between virtual and non-virtual methods:

```

31     using System;
32     class A
33     {
34         public void F() { Console.WriteLine("A.F"); }
35         public virtual void G() { Console.WriteLine("A.G"); }
36     }
37     class B: A
38     {
39         new public void F() { Console.WriteLine("B.F"); }
40         public override void G() { Console.WriteLine("B.G"); }
41     }
  
```



```

1      class Test
2      {
3          static void Main() {
4              B b = new B();
5              A a = b;
6              a.F();
7              b.F();
8              a.G();
9              b.G();
10         }
11     }

```

12 In the example, A introduces a non-virtual method F and a virtual method G. The class B introduces a *new*
 13 non-virtual method F, thus *hiding* the inherited F, and also *overrides* the inherited method G. The example
 14 produces the output:

```

15     A.F
16     B.F
17     B.G
18     B.G

```

19 Notice that the statement `a.G()` invokes `B.G`, not `A.G`. This is because the run-time type of the instance
 20 (which is B), not the compile-time type of the instance (which is A), determines the actual method
 21 implementation to invoke. *end example*]

22 Because methods are allowed to hide inherited methods, it is possible for a class to contain several virtual
 23 methods with the same signature. This does not present an ambiguity problem, since all but the most derived
 24 method are hidden. [*Example*: In the example

```

25     using System;
26     class A
27     {
28         public virtual void F() { Console.WriteLine("A.F"); }
29     }
30     class B: A
31     {
32         public override void F() { Console.WriteLine("B.F"); }
33     }
34     class C: B
35     {
36         new public virtual void F() { Console.WriteLine("C.F"); }
37     }
38     class D: C
39     {
40         public override void F() { Console.WriteLine("D.F"); }
41     }
42     class Test
43     {
44         static void Main() {
45             D d = new D();
46             A a = d;
47             B b = d;
48             C c = d;
49             a.F();
50             b.F();
51             c.F();
52             d.F();
53         }
54     }

```

55 the C and D classes contain two virtual methods with the same signature: The one introduced by A and the
 56 one introduced by C. The method introduced by C hides the method inherited from A. Thus, the override
 57 declaration in D overrides the method introduced by C, and it is not possible for D to override the method
 58 introduced by A. The example produces the output:

```

1      B.F
2      B.F
3      D.F
4      D.F

```

5 Note that it is possible to invoke the hidden virtual method by accessing an instance of D through a less
6 derived type in which the method is not hidden. *end example*]

7 **17.5.4 Override methods**

8 When an instance method declaration includes an **override** modifier, the method is said to be an *override*
9 *method*. An override method overrides an inherited virtual method with the same signature. Whereas a
10 virtual method declaration *introduces* a new method, an override method declaration *specializes* an existing
11 inherited virtual method by providing a new implementation of that method.

12 The method overridden by an **override** declaration is known as the *overridden base method*. For an
13 override method M declared in a class C, the overridden base method is determined by examining each base
14 class of C, starting with the direct base class of C and continuing with each successive direct base class, until
15 an accessible method with the same signature as M is located. For the purposes of locating the overridden
16 base method, a method is considered accessible if it is **public**, if it is **protected**, if it is **protected**
17 **internal**, or if it is **internal** and declared in the same program as C.

18 A compile-time error occurs unless all of the following are true for an override declaration:

- 19 • An overridden base method can be located as described above.
- 20 • The overridden base method is a virtual, abstract, or override method. In other words, the overridden
21 base method cannot be static or non-virtual.
- 22 • The overridden base method is not a sealed method.
- 23 • The override declaration and the overridden base method have the same declared accessibility. In other
24 words, an override declaration cannot change the accessibility of the virtual method.

25 An override declaration can access the overridden base method using a *base-access* (§14.5.8). [*Example*: In
26 the example

```

27     class A
28     {
29         int x;
30         public virtual void PrintFields() {
31             Console.WriteLine("x = {0}", x);
32         }
33     }
34     class B: A
35     {
36         int y;
37         public override void PrintFields() {
38             base.PrintFields();
39             Console.WriteLine("y = {0}", y);
40         }
41     }

```

42 the `base.PrintFields()` invocation in B invokes the `PrintFields` method declared in A. A *base-*
43 *access* disables the virtual invocation mechanism and simply treats the base method as a non-virtual method.
44 Had the invocation in B been written `((A)this).PrintFields()`, it would recursively invoke the
45 `PrintFields` method declared in B, not the one declared in A, since `PrintFields` is virtual and the run-
46 time type of `((A)this)` is B. *end example*]

47 Only by including an **override** modifier can a method override another method. In all other cases, a
48 method with the same signature as an inherited method simply hides the inherited method. [*Example*: In the
49 example

```

1     class A
2     {
3         public virtual void F() {}
4     }
5     class B: A
6     {
7         public virtual void F() {}    // warning, hiding inherited F()
8     }

```

9 the F method in B does not include an **override** modifier and therefore does not override the F method
10 in A. Rather, the F method in B hides the method in A, and a warning is reported because the declaration does
11 not include a **new** modifier. *end example*]

12 [*Example:* In the example

```

13     class A
14     {
15         public virtual void F() {}
16     }
17     class B: A
18     {
19         new private void F() {}      // Hides A.F within B
20     }
21     class C: B
22     {
23         public override void F() {} // Ok, overrides A.F
24     }

```

25 the F method in B hides the virtual F method inherited from A. Since the new F in B has private access, its
26 scope only includes the class body of B and does not extend to C. Therefore, the declaration of F in C is
27 permitted to override the F inherited from A. *end example*]

28 17.5.5 Sealed methods

29 When an instance method declaration includes a **sealed** modifier, that method is said to be a **sealed**
30 **method**. A sealed method overrides an inherited virtual method with the same signature. An override method
31 can also be marked with the **sealed** modifier. Use of this modifier prevents a derived class from further
32 overriding the method.

33 [*Example:* The example

```

34     using System;
35     class A
36     {
37         public virtual void F() {
38             Console.WriteLine("A.F");
39         }
40         public virtual void G() {
41             Console.WriteLine("A.G");
42         }
43     }
44     class B: A
45     {
46         sealed override public void F() {
47             Console.WriteLine("B.F");
48         }
49         override public void G() {
50             Console.WriteLine("B.G");
51         }
52     }

```

```

1      class C: B
2      {
3          override public void G() {
4              Console.WriteLine("C.G");
5          }
6      }

```

7 the class B provides two override methods: an F method that has the `sealed` modifier and a G method that
8 does not. B's use of the `sealed` modifier prevents C from further overriding F. *end example*

9 17.5.6 Abstract methods

10 When an instance method declaration includes an `abstract` modifier, that method is said to be an *abstract*
11 *method*. Although an abstract method is implicitly also a virtual method, it cannot have the modifier
12 `virtual`.

13 An abstract method declaration introduces a new virtual method but does not provide an implementation of
14 that method. Instead, non-abstract derived classes are required to provide their own implementation by
15 overriding that method. Because an abstract method provides no actual implementation, the *method-body* of
16 an abstract method simply consists of a semicolon.

17 Abstract method declarations are only permitted in abstract classes (§17.1.1.1).

18 [*Example:* In the example

```

19      public abstract class Shape
20      {
21          public abstract void Paint(Graphics g, Rectangle r);
22      }
23
24      public class Ellipse: Shape
25      {
26          public override void Paint(Graphics g, Rectangle r) {
27              g.DrawEllipse(r);
28          }
29
30          public class Box: Shape
31          {
32              public override void Paint(Graphics g, Rectangle r) {
33                  g.DrawRect(r);
34              }
35          }
36      }

```

35 the Shape class defines the abstract notion of a geometrical shape object that can paint itself. The Paint
36 method is abstract because there is no meaningful default implementation. The Ellipse and Box classes
37 are concrete Shape implementations. Because these classes are non-abstract, they are required to override
38 the Paint method and provide an actual implementation. *end example*

39 It is a compile-time error for a *base-access* (§14.5.8) to reference an abstract method. [*Example:* In the
40 example

```

41      abstract class A
42      {
43          public abstract void F();
44      }
45
46      class B: A
47      {
48          public override void F() {
49              base.F();           // Error, base.F is abstract
50          }
51      }

```

51 a compile-time error is reported for the `base.F()` invocation because it references an abstract method. *end*
52 *example*

1 An abstract method declaration is permitted to override a virtual method. This allows an abstract class to
 2 force re-implementation of the method in derived classes, and makes the original implementation of the
 3 method unavailable. [Example: In the example

```

4     using System;
5     class A
6     {
7         public virtual void F() {
8             Console.WriteLine("A.F");
9         }
10    }
11    abstract class B: A
12    {
13        public abstract override void F();
14    }
15    class C: B
16    {
17        public override void F() {
18            Console.WriteLine("C.F");
19        }
20    }

```

21 class A declares a virtual method, class B overrides this method with an abstract method, and class C
 22 overrides that abstract method to provide its own implementation. *end example*]

23 17.5.7 External methods

24 When a method declaration includes an `extern` modifier, the method is said to be an *external method*.
 25 External methods are implemented externally, typically using a language other than C#. Because an external
 26 method declaration provides no actual implementation, the *method-body* of an external method simply
 27 consists of a semicolon.

28 The mechanism by which linkage to an external method is achieved, is implementation-defined.

29 [Example: The following example demonstrates the use of the `extern` modifier in combination with a
 30 `DllImport` attribute that specifies the name of the external library in which the method is implemented:

```

31     using System.Text;
32     using System.Security.Permissions;
33     using System.Runtime.InteropServices;
34     class Path
35     {
36         [DllImport("kernel32", SetLastError=true)]
37         static extern bool CreateDirectory(string name, SecurityAttribute sa);
38         [DllImport("kernel32", SetLastError=true)]
39         static extern bool RemoveDirectory(string name);
40         [DllImport("kernel32", SetLastError=true)]
41         static extern int GetCurrentDirectory(int bufSize, StringBuilder buf);
42         [DllImport("kernel32", SetLastError=true)]
43         static extern bool SetCurrentDirectory(string name);
44     }

```

45 *end example*]

46 17.5.8 Method body

47 The *method-body* of a method declaration consists of either a *block* or a semicolon.

48 Abstract and external method declarations do not provide a method implementation, so their method bodies
 49 simply consist of a semicolon. For any other method, the method body is a block (§15.2) that contains the
 50 statements to execute when that method is invoked.

1 When the return type of a method is `void`, `return` statements (§15.9.4) in that method's body are not
 2 permitted to specify an expression. If execution of the method body of a `void` method completes normally
 3 (that is, control flows off the end of the method body), that method simply returns to its caller.

4 When the return type of a method is not `void`, each `return` statement in that method body must specify an
 5 expression of a type that is implicitly convertible to the return type. The endpoint of the method body of a
 6 value-returning method must not be reachable. In other words, in a value-returning method, control is not
 7 permitted to flow off the end of the method body.

8 [*Example:* In the example

```

 9     class A
10     {
11         public int F() {}           // Error, return value required
12         public int G() {
13             return 1;
14         }
15         public int H(bool b) {
16             if (b) {
17                 return 1;
18             }
19             else {
20                 return 0;
21             }
22         }
23     }

```

24 the value-returning `F` method results in a compile-time error because control can flow off the end of the
 25 method body. The `G` and `H` methods are correct because all possible execution paths end in a `return`
 26 statement that specifies a return value. *end example*]

27 17.5.9 Method overloading

28 The method overload resolution rules are described in §14.4.2.

29 17.6 Properties

30 A *property* is a member that provides access to an attribute of an object or a class. Examples of properties
 31 include the length of a string, the size of a font, the caption of a window, the name of a customer, and so on.
 32 Properties are a natural extension of fields—both are named members with associated types, and the syntax
 33 for accessing fields and properties is the same. However, unlike fields, properties do not denote storage
 34 locations. Instead, properties have *accessors* that specify the statements to be executed when their values are
 35 read or written. Properties thus provide a mechanism for associating actions with the reading and writing of
 36 an object's attributes; furthermore, they permit such attributes to be computed.

37 Properties are declared using *property-declarations*:

```

38     property-declaration:
39         attributesopt property-modifiersopt type member-name { accessor-declarations }
40
41     property-modifiers:
42         property-modifier
43         property-modifiers property-modifier

```

```

1      property-modifier:
2          new
3          public
4          protected
5          internal
6          private
7          static
8          virtual
9          sealed
10         override
11         abstract
12         extern
13
14     member-name:
15         identifier
16         interface-type . identifier

```

16 A *property-declaration* may include a set of *attributes* (§24) and a valid combination of the four access
17 modifiers (§17.2.3), the `new` (§17.2.2), `static` (§17.6.1), `virtual` (§17.5.3, §17.6.3), `override` (§17.5.4,
18 §17.6.3), `sealed` (§17.5.5), `abstract` (§17.5.6, §17.6.3), and `extern` modifiers.

19 Property declarations are subject to the same rules as method declarations (§17.5) with regard to valid
20 combinations of modifiers.

21 The *type* of a property declaration specifies the type of the property introduced by the declaration, and the
22 *member-name* specifies the name of the property. Unless the property is an explicit interface member
23 implementation, the *member-name* is simply an *identifier*. For an explicit interface member implementation
24 (§20.4.1), the *member-name* consists of an *interface-type* followed by a “.” and an *identifier*.

25 The *type* of a property must be at least as accessible as the property itself (§10.5.4).

26 The *accessor-declarations*, which must be enclosed in “{” and “}” tokens, declare the accessors (§17.6.2) of
27 the property. The accessors specify the executable statements associated with reading and writing the
28 property.

29 Even though the syntax for accessing a property is the same as that for a field, a property is not classified as
30 a variable. Thus, it is not possible to pass a property as a `ref` or `out` argument.

31 When a property declaration includes an `extern` modifier, the property is said to be an *external property*.
32 Because an external property declaration provides no actual implementation, each of its *accessor-*
33 *declarations* consists of a semicolon.

34 17.6.1 Static and instance properties

35 When a property declaration includes a `static` modifier, the property is said to be a *static property*. When
36 no `static` modifier is present, the property is said to be an *instance property*.

37 A static property is not associated with a specific instance, and it is a compile-time error to refer to `this` in
38 the accessors of a static property.

39 An instance property is associated with a given instance of a class, and that instance can be accessed as
40 `this` (§14.5.7) in the accessors of that property.

41 When a property is referenced in a *member-access* (§14.5.4) of the form `E.M`, if `M` is a static property, `E` must
42 denote a type that has a property `M`, and if `M` is an instance property, `E` must denote an instance having a
43 property `M`.

44 The differences between static and instance members are discussed further in §17.2.5.

1 **17.6.2 Accessors**

2 The *accessor-declarations* of a property specify the executable statements associated with reading and
3 writing that property.

```
4     accessor-declarations:
5         get-accessor-declaration set-accessor-declarationopt
6         set-accessor-declaration get-accessor-declarationopt
7
8     get-accessor-declaration:
9         attributesopt get accessor-body
10
11    set-accessor-declaration:
12        attributesopt set accessor-body
13
14    accessor-body:
15        block
16    ;
```

14 The accessor declarations consist of a *get-accessor-declaration*, a *set-accessor-declaration*, or both. Each
15 accessor declaration consists of the token **get** or **set** followed by an *accessor-body*. For **abstract** and
16 **extern** properties, the *accessor-body* for each accessor specified is simply a semicolon. For the accessors
17 of any non-abstract, non-extern property, the *accessor-body* is a *block* which specifies the statements to be
18 executed when the corresponding accessor is invoked.

19 A **get** accessor corresponds to a parameterless method with a return value of the property type. Except as
20 the target of an assignment, when a property is referenced in an expression, the **get** accessor of the property
21 is invoked to compute the value of the property (§14.1.1). The body of a **get** accessor must conform to the
22 rules for value-returning methods described in §17.5.8. In particular, all **return** statements in the body of a
23 **get** accessor must specify an expression that is implicitly convertible to the property type. Furthermore, the
24 endpoint of a **get** accessor must not be reachable.

25 A **set** accessor corresponds to a method with a single value parameter of the property type and a **void**
26 return type. The implicit parameter of a **set** accessor is always named **value**. When a property is
27 referenced as the target of an assignment (§14.13), or as the operand of **++** or **--** (§14.5.9, 14.6.5), the **set**
28 accessor is invoked with an argument (whose value is that of the right-hand side of the assignment or the
29 operand of the **++** or **--** operator) that provides the new value (§14.13.1). The body of a **set** accessor must
30 conform to the rules for **void** methods described in §17.5.8. In particular, **return** statements in the **set**
31 accessor body are not permitted to specify an expression. Since a **set** accessor implicitly has a parameter
32 named **value**, it is a compile-time error for a local variable declaration in a **set** accessor to have that name.

33 Based on the presence or absence of the **get** and **set** accessors, a property is classified as follows:

- 34 • A property that includes both a **get** accessor and a **set** accessor is said to be a **read-write** property.
- 35 • A property that has only a **get** accessor is said to be a **read-only** property. It is a compile-time error for
36 a read-only property to be the target of an assignment.
- 37 • A property that has only a **set** accessor is said to be a **write-only** property. Except as the target of an
38 assignment, it is a compile-time error to reference a write-only property in an expression. [*Note*: The
39 pre- and postfix **++** and **--** operators cannot be applied to write-only properties, since these operators
40 read the old value of their operand before they write the new one. *end note*]

41 [*Example*: In the example

```
42     public class Button: Control
43     {
44         private string caption;
```



```

1      public string Caption {
2          get {
3              return caption;
4          }
5          set {
6              if (caption != value) {
7                  caption = value;
8                  Repaint();
9              }
10         }
11     }
12     public override void Paint(Graphics g, Rectangle r) {
13         // Painting code goes here
14     }
15 }

```

16 the `Button` control declares a public `Caption` property. The `get` accessor of the `Caption` property returns
17 the string stored in the private `caption` field. The `set` accessor checks if the new value is different from the
18 current value, and if so, it stores the new value and repaints the control. Properties often follow the pattern
19 shown above: The `get` accessor simply returns a value stored in a private field, and the `set` accessor
20 modifies that private field and then performs any additional actions required to fully update the state of the
21 object.

22 Given the `Button` class above, the following is an example of use of the `Caption` property:

```

23     Button okButton = new Button();
24     okButton.Caption = "OK";           // Invokes set accessor
25     string s = okButton.Caption;      // Invokes get accessor

```

26 Here, the `set` accessor is invoked by assigning a value to the property, and the `get` accessor is invoked by
27 referencing the property in an expression. *end example*

28 The `get` and `set` accessors of a property are not distinct members, and it is not possible to declare the
29 accessors of a property separately. [*Note*: As such, it is not possible for the two accessors of a read-write
30 property to have different accessibility. *end note*] [*Example*: The example

```

31     class A
32     {
33         private string name;
34         public string Name {           // Error, duplicate member name
35             get { return name; }
36         }
37         public string Name {           // Error, duplicate member name
38             set { name = value; }
39         }
40     }

```

41 does not declare a single read-write property. Rather, it declares two properties with the same name, one
42 read-only and one write-only. Since two members declared in the same class cannot have the same name, the
43 example causes a compile-time error to occur. *end example*

44 When a derived class declares a property by the same name as an inherited property, the derived property
45 hides the inherited property with respect to both reading and writing. [*Example*: In the example

```

46     class A
47     {
48         public int P {
49             set {...}
50         }
51     }
52     class B: A
53     {
54         new public int P {
55             get {...}
56         }
57     }

```

C# LANGUAGE SPECIFICATION

1 the P property in B hides the P property in A with respect to both reading and writing. Thus, in the
2 statements

```
3     B b = new B();  
4     b.P = 1; // Error, B.P is read-only  
5     ((A)b).P = 1; // Ok, reference to A.P
```

6 the assignment to `b.P` causes a compile-time error to be reported, since the read-only P property in B hides
7 the write-only P property in A. Note, however, that a cast can be used to access the hidden P property. *end*
8 *example*]

9 Unlike public fields, properties provide a separation between an object's internal state and its public
10 interface. [*Example*: Consider the example:

```
11     class Label  
12     {  
13         private int x, y;  
14         private string caption;  
15         public Label(int x, int y, string caption) {  
16             this.x = x;  
17             this.y = y;  
18             this.caption = caption;  
19         }  
20         public int X {  
21             get { return x; }  
22         }  
23         public int Y {  
24             get { return y; }  
25         }  
26         public Point Location {  
27             get { return new Point(x, y); }  
28         }  
29         public string Caption {  
30             get { return caption; }  
31         }  
32     }
```

33 Here, the `Label` class uses two `int` fields, `x` and `y`, to store its location. The location is publicly exposed
34 both as an `X` and a `Y` property and as a `Location` property of type `Point`. If, in a future version of `Label`,
35 it becomes more convenient to store the location as a `Point` internally, the change can be made without
36 affecting the public interface of the class:

```
37     class Label  
38     {  
39         private Point location;  
40         private string caption;  
41         public Label(int x, int y, string caption) {  
42             this.location = new Point(x, y);  
43             this.caption = caption;  
44         }  
45         public int X {  
46             get { return location.x; }  
47         }  
48         public int Y {  
49             get { return location.y; }  
50         }  
51         public Point Location {  
52             get { return location; }  
53         }  
54         public string Caption {  
55             get { return caption; }  
56         }  
57     }
```

1 Had `x` and `y` instead been `public readonly` fields, it would have been impossible to make such a change
2 to the `Label` class. *end example*

3 [*Note:* Exposing state through properties is not necessarily any less efficient than exposing fields directly. In
4 particular, when a property is non-virtual and contains only a small amount of code, the execution
5 environment may replace calls to accessors with the actual code of the accessors. This process is known as
6 **inlining**, and it makes property access as efficient as field access, yet preserves the increased flexibility of
7 properties. *end note*]

8 [*Example:* Since invoking a `get` accessor is conceptually equivalent to reading the value of a field, it is
9 considered bad programming style for `get` accessors to have observable side-effects. In the example

```
10     class Counter
11     {
12         private int next;
13         public int Next {
14             get { return next++; }
15         }
16     }
```

17 the value of the `Next` property depends on the number of times the property has previously been accessed.
18 Thus, accessing the property produces an observable side effect, and the property should be implemented as
19 a method instead. *end example*

20 [*Note:* The “no side-effects” convention for `get` accessors doesn’t mean that `get` accessors should always
21 be written to simply return values stored in fields. Indeed, `get` accessors often compute the value of a
22 property by accessing multiple fields or invoking methods. However, a properly designed `get` accessor
23 performs no actions that cause observable changes in the state of the object. *end note*]

24 Properties can be used to delay initialization of a resource until the moment it is first referenced. [*Example:*
25 For example:

```
26     using System.IO;
27     public class Console
28     {
29         private static TextReader reader;
30         private static TextWriter writer;
31         private static TextWriter error;
32         public static TextReader In {
33             get {
34                 if (reader == null) {
35                     reader = new StreamReader(Console.OpenStandardInput());
36                 }
37                 return reader;
38             }
39         }
40         public static TextWriter Out {
41             get {
42                 if (writer == null) {
43                     writer = new StreamWriter(Console.OpenStandardOutput());
44                 }
45                 return writer;
46             }
47         }
48         public static TextWriter Error {
49             get {
50                 if (error == null) {
51                     error = new StreamWriter(Console.OpenStandardError());
52                 }
53                 return error;
54             }
55         }
56     }
```

1 The `Console` class contains three properties, `In`, `Out`, and `Error`, that represent the standard input, output,
2 and error devices, respectively. By exposing these members as properties, the `Console` class can delay their
3 initialization until they are actually used. For example, upon first referencing the `Out` property, as in

```
4     Console.Out.WriteLine("hello, world");
```

5 the underlying `TextWriter` for the output device is created. But if the application makes no reference to the
6 `In` and `Error` properties, then no objects are created for those devices. *end example*]

7 **17.6.3 Virtual, sealed, override, and abstract accessors**

8 A `virtual` property declaration specifies that the accessors of the property are virtual. The `virtual`
9 modifier applies to both accessors of a read-write property—it is not possible for only one accessor of a
10 read-write property to be virtual.

11 An `abstract` property declaration specifies that the accessors of the property are virtual, but does not
12 provide an actual implementation of the accessors. Instead, non-abstract derived classes are required to
13 provide their own implementation for the accessors by overriding the property. Because an accessor for an
14 abstract property declaration provides no actual implementation, its *accessor-body* simply consists of a
15 semicolon.

16 A property declaration that includes both the `abstract` and `override` modifiers specifies that the property
17 is abstract and overrides a base property. The accessors of such a property are also abstract.

18 Abstract property declarations are only permitted in abstract classes (§17.1.1.1). The accessors of an
19 inherited virtual property can be overridden in a derived class by including a property declaration that
20 specifies an `override` directive. This is known as an *overriding property declaration*. An overriding
21 property declaration does not declare a new property. Instead, it simply specializes the implementations of
22 the accessors of an existing virtual property.

23 An overriding property declaration must specify the exact same accessibility modifiers, type, and name as
24 the inherited property. If the inherited property has only a single accessor (i.e., if the inherited property is
25 read-only or write-only), the overriding property must include only that accessor. If the inherited property
26 includes both accessors (i.e., if the inherited property is read-write), the overriding property can include
27 either a single accessor or both accessors.

28 An overriding property declaration may include the `sealed` modifier. Use of this modifier prevents a
29 derived class from further overriding the property. The accessors of a sealed property are also sealed.

30 Except for differences in declaration and invocation syntax, `virtual`, `sealed`, `override`, and `abstract` accessors
31 behave exactly like `virtual`, `sealed`, `override` and `abstract` methods. Specifically, the rules described in
32 §17.5.3, §17.5.4, §17.5.5, and §17.5.6 apply as if accessors were methods of a corresponding form:

- 33 • A `get` accessor corresponds to a parameterless method with a return value of the property type and the
34 same modifiers as the containing property.
- 35 • A `set` accessor corresponds to a method with a single value parameter of the property type, a `void`
36 return type, and the same modifiers as the containing property.

37 [*Example:* In the example

```
38     abstract class A
39     {
40         int y;
41         public virtual int x {
42             get { return 0; }
43         }
44         public virtual int Y {
45             get { return y; }
46             set { y = value; }
47         }
48         public abstract int Z { get; set; }
49     }
```

1 X is a virtual read-only property, Y is a virtual read-write property, and Z is an abstract read-write property.
 2 Because Z is abstract, the containing class A must also be declared abstract.

3 A class that derives from A is show below:

```

4     class B: A
5     {
6         int z;
7         public override int X {
8             get { return base.X + 1; }
9         }
10        public override int Y {
11            set { base.Y = value < 0? 0: value; }
12        }
13        public override int Z {
14            get { return z; }
15            set { z = value; }
16        }
17    }

```

18 Here, the declarations of X, Y, and Z are overriding property declarations. Each property declaration exactly
 19 matches the accessibility modifiers, type, and name of the corresponding inherited property. The `get`
 20 accessor of X and the `set` accessor of Y use the `base` keyword to access the inherited accessors. The
 21 declaration of Z overrides both abstract accessors—thus, there are no outstanding abstract function members
 22 in B, and B is permitted to be a non-abstract class. *end example*]

23 17.7 Events

24 An *event* is a member that enables an object or class to provide notifications. Clients can attach executable
 25 code for events by supplying *event handlers*.

26 Events are declared using *event-declarations*:

```

27     event-declaration:
28         attributesopt event-modifiersopt event type variable-declarators ;
29         attributesopt event-modifiersopt event type member-name { event-accessor-declarations
30     }

31     event-modifiers:
32         event-modifier
33         event-modifiers event-modifier

34     event-modifier:
35         new
36         public
37         protected
38         internal
39         private
40         static
41         virtual
42         sealed
43         override
44         abstract
45         extern

46     event-accessor-declarations:
47         add-accessor-declaration remove-accessor-declaration
48         remove-accessor-declaration add-accessor-declaration

49     add-accessor-declaration:
50         attributesopt add block

```

C# LANGUAGE SPECIFICATION

1 *remove-accessor-declaration:*
2 *attributes*_{opt} **remove** *block*

3 An *event-declaration* may include a set of *attributes* (§24) and a valid combination of the four access
4 modifiers (§17.2.3), the **new** (§17.2.2), **static** (§17.5.2, §17.7.3), **virtual** (§17.5.3, §17.7.4), **override**
5 (§17.5.4, §17.7.4), **sealed** (§17.5.5), **abstract** (§17.5.6, §17.7.4), and **extern** modifiers.

6 Event declarations are subject to the same rules as method declarations (§17.5) with regard to valid
7 combinations of modifiers.

8 The *type* of an event declaration must be a *delegate-type* (§11.2), and that *delegate-type* must be at least as
9 accessible as the event itself (§10.5.4).

10 An event declaration may include *event-accessor-declarations*. However, if it does not, for non-extern, non-
11 abstract events, the compiler shall supply them automatically (§17.7.1); for extern events, the accessors are
12 provided externally.

13 An event declaration that omits *event-accessor-declarations* defines one or more events—one for each of the
14 *variable-declarators*. The attributes and modifiers apply to all of the members declared by such an *event-*
15 *declaration*.

16 It is a compile-time error for an *event-declaration* to include both the **abstract** modifier and brace-
17 delimited *event-accessor-declarations*.

18 When an event declaration includes an **extern** modifier, the event is said to be an *external event*. Because
19 an external event declaration provides no actual implementation, it is an error for it to include both the
20 **extern** modifier and *event-accessor-declarations*.

21 An event can be used as the left-hand operand of the **+=** and **-=** operators (§14.13.3). These operators are
22 used, respectively, to attach event handlers to, or to remove event handlers from an event, and the access
23 modifiers of the event control the contexts in which such operations are permitted.

24 Since **+=** and **-=** are the only operations that are permitted on an event outside the type that declares the
25 event, external code can add and remove handlers for an event, but cannot in any other way obtain or modify
26 the underlying list of event handlers.

27 In an operation of the form **x += y** or **x -= y**, when **x** is an event and the reference takes place outside the
28 type that contains the declaration of **x**, the result of the operation has type **void** (as opposed to having the
29 type of **x**, with the value of **x** after the assignment). This rule prohibits external code from indirectly
30 examining the underlying delegate of an event.

31 [*Example:* The following example shows how event handlers are attached to instances of the **Button** class:

```
32     public delegate void EventHandler(object sender, EventArgs e);  
33     public class Button: Control  
34     {  
35         public event EventHandler Click;  
36     }  
37     public class LoginDialog: Form  
38     {  
39         Button OkButton;  
40         Button CancelButton;  
41         public LoginDialog() {  
42             OkButton = new Button(...);  
43             OkButton.Click += new EventHandler(OkButtonClick);  
44             CancelButton = new Button(...);  
45             CancelButton.Click += new EventHandler(CancelButtonClick);  
46         }  
47         void OkButtonClick(object sender, EventArgs e) {  
48             // Handle OkButton.Click event  
49         }
```

```

1         void CancelButtonClick(object sender, EventArgs e) {
2             // Handle CancelButton.Click event
3         }
4     }

```

5 Here, the `LoginDialog` instance constructor creates two `Button` instances and attaches event handlers to
6 the `Click` events. *end example*]

7 17.7.1 Field-like events

8 Within the program text of the class or struct that contains the declaration of an event, certain events can be
9 used like fields. To be used in this way, an event must not be `abstract` or `extern`, and must not explicitly
10 include *event-accessor-declarations*. Such an event can be used in any context that permits a field. The field
11 contains a delegate (§22), which refers to the list of event handlers that have been added to the event. If no
12 event handlers have been added, the field contains `null`.

13 [*Example:* In the example

```

14         public delegate void EventHandler(object sender, EventArgs e);
15         public class Button: Control
16         {
17             public event EventHandler Click;
18             protected void OnClick(EventArgs e) {
19                 if (Click != null) Click(this, e);
20             }
21             public void Reset() {
22                 Click = null;
23             }
24         }

```

25 `Click` is used as a field within the `Button` class. As the example demonstrates, the field can be examined,
26 modified, and used in delegate invocation expressions. The `OnClick` method in the `Button` class “raises”
27 the `Click` event. The notion of raising an event is precisely equivalent to invoking the delegate represented
28 by the event—thus, there are no special language constructs for raising events. Note that the delegate
29 invocation is preceded by a check that ensures the delegate is non-null.

30 Outside the declaration of the `Button` class, the `Click` member can only be used on the left-hand side of
31 the `+=` and `-=` operators, as in

```
32         b.Click += new EventHandler(...);
```

33 which appends a delegate to the invocation list of the `Click` event, and

```
34         b.Click -= new EventHandler(...);
```

35 which removes a delegate from the invocation list of the `Click` event. *end example*]

36 When compiling a field-like event, the compiler automatically creates storage to hold the delegate, and
37 creates accessors for the event that add or remove event handlers to the delegate field. In order to be thread-
38 safe, the addition or removal operations are done while holding the lock (§15.12) on the containing object
39 for an instance event, or the type object (§14.5.11) for a static event.

40 [*Note:* Thus, an instance event declaration of the form:

```

41         class X {
42             public event D Ev;
43         }

```

44 could be compiled to something equivalent to:

```

45         class X {
46             private D __Ev; // field to hold the delegate
47             public event D Ev {
48                 add {
49                     lock(this) { __Ev = __Ev + value; }
50                 }

```

```

1         remove {
2             lock(this) { __Ev = __Ev - value; }
3         }
4     }
5 }

```

6 Within the class X, references to EV are compiled to reference the hidden field __EV instead. The name
7 “__EV” is arbitrary; the hidden field could have any name or no name at all.

8 Similarly, a static event declaration of the form:

```

9     class X {
10         public static event D EV;
11     }

```

12 could be compiled to something equivalent to:

```

13     class X {
14         private static D __Ev; // field to hold the delegate
15         public static event D EV {
16             add {
17                 lock(typeof(X)) { __Ev = __Ev + value; }
18             }
19             remove {
20                 lock(typeof(X)) { __Ev = __Ev - value; }
21             }
22         }
23     }

```

24 *end note]*

25 17.7.2 Event accessors

26 [*Note:* Event declarations typically omit *event-accessor-declarations*, as in the `Button` example above. One
27 situation for doing so involves the case in which the storage cost of one field per event is not acceptable. In
28 such cases, a class can include *event-accessor-declarations* and use a private mechanism for storing the list
29 of event handlers. Similarly, in cases where the handling of an event requires access to external resources,
30 event accessors may be used to manage these resources. *end note]*

31 The *event-accessor-declarations* of an event specify the executable statements associated with adding and
32 removing event handlers.

33 The accessor declarations consist of an *add-accessor-declaration* and a *remove-accessor-declaration*. Each
34 accessor declaration consists of the token `add` or `remove` followed by a *block*. The *block* associated with an
35 *add-accessor-declaration* specifies the statements to execute when an event handler is added, and the *block*
36 associated with a *remove-accessor-declaration* specifies the statements to execute when an event handler is
37 removed.

38 Each *add-accessor-declaration* and *remove-accessor-declaration* corresponds to a method with a single
39 value parameter of the event type, and a `void` return type. The implicit parameter of an event accessor is
40 named `value`. When an event is used in an event assignment, the appropriate event accessor is used.
41 Specifically, if the assignment operator is `+=` then the add accessor is used, and if the assignment operator is
42 `-=` then the remove accessor is used. In either case, the right-hand operand of the assignment operator is
43 used as the argument to the event accessor. The block of an *add-accessor-declaration* or a *remove-accessor-*
44 *declaration* must conform to the rules for `void` methods described in §17.5.8. In particular, `return`
45 statements in such a block are not permitted to specify an expression.

46 Since an event accessor implicitly has a parameter named `value`, it is a compile-time error for a local
47 variable declared in an event accessor to have that name.

48 [*Example:* In the example


```

1  class Control: Component
2  {
3      // Unique keys for events
4      static readonly object mouseDownEventKey = new object();
5      static readonly object mouseUpEventKey = new object();
6
7      // Return event handler associated with key
8      protected Delegate GetEventHandler(object key) {...}
9
10     // Add event handler associated with key
11     protected void AddEventHandler(object key, Delegate handler) {...}
12
13     // Remove event handler associated with key
14     protected void RemoveEventHandler(object key, Delegate handler) {...}
15
16     // MouseDown event
17     public event EventHandler MouseDown {
18         add { AddEventHandler(mouseDownEventKey, value); }
19         remove { RemoveEventHandler(mouseDownEventKey, value); }
20     }
21
22     // MouseUp event
23     public event EventHandler MouseUp {
24         add { AddEventHandler(mouseUpEventKey, value); }
25         remove { RemoveEventHandler(mouseUpEventKey, value); }
26     }
27
28     // Invoke the MouseUp event
29     protected void OnMouseUp(MouseEventArgs args) {
30         EventHandler handler;
31         handler = (EventHandler)GetEventHandler(mouseUpEventKey);
32         if (handler != null)
33             handler(this, args);
34     }
35 }

```

30 the `Control` class implements an internal storage mechanism for events. The `AddEventHandler` method
31 associates a delegate value with a key, the `GetEventHandler` method returns the delegate currently
32 associated with a key, and the `RemoveEventHandler` method removes a delegate as an event handler for
33 the specified event. Presumably, the underlying storage mechanism is designed such that there is no cost for
34 associating a `null` delegate value with a key, and thus unhandled events consume no storage. *end example*]

35 17.7.3 Static and instance events

36 When an event declaration includes a `static` modifier, the event is said to be a *static event*. When no
37 `static` modifier is present, the event is said to be an *instance event*.

38 A static event is not associated with a specific instance, and it is a compile-time error to refer to `this` in the
39 accessors of a static event.

40 An instance event is associated with a given instance of a class, and this instance can be accessed as `this`
41 (§14.5.7) in the accessors of that event.

42 When an event is referenced in a *member-access* (§14.5.4) of the form `E.M`, if `M` is a static event, `E` must
43 denote a type, and if `M` is an instance event, `E` must denote an instance.

44 The differences between static and instance members are discussed further in §17.2.5.

45 17.7.4 Virtual, sealed, override, and abstract accessors

46 A `virtual` event declaration specifies that the accessors of that event are virtual. The `virtual` modifier
47 applies to both accessors of an event.

48 An `abstract` event declaration specifies that the accessors of the event are virtual, but does not provide an
49 actual implementation of the accessors. Instead, non-abstract derived classes are required to provide their
50 own implementation for the accessors by overriding the event. Because an accessor for an abstract event
51 declaration provides no actual implementation, its *accessor-body* simply consists of a semicolon.

1 An event declaration that includes both the **abstract** and **override** modifiers specifies that the event is
2 abstract and overrides a base event. The accessors of such an event are also abstract.

3 Abstract event declarations are only permitted in abstract classes (§17.1.1.1).

4 The accessors of an inherited virtual event can be overridden in a derived class by including an event
5 declaration that specifies an **override** modifier. This is known as an **overriding event declaration**. An
6 overriding event declaration does not declare a new event. Instead, it simply specializes the implementations
7 of the accessors of an existing virtual event.

8 An overriding event declaration must specify the exact same accessibility modifiers, type, and name as the
9 overridden event.

10 An overriding event declaration may include the **sealed** modifier. Use of this modifier prevents a derived
11 class from further overriding the event. The accessors of a sealed event are also sealed.

12 It is a compile-time error for an overriding event declaration to include a **new** modifier.

13 Except for differences in declaration and invocation syntax, virtual, sealed, override, and abstract accessors
14 behave exactly like virtual, sealed, override and abstract methods. Specifically, the rules described in
15 §17.5.3, §17.5.4, §17.5.5, and §17.5.6 apply as if accessors were methods of a corresponding form. Each
16 accessor corresponds to a method with a single value parameter of the event type, a void return type, and
17 the same modifiers as the containing event.

18 17.8 Indexers

19 An **indexer** is a member that enables an object to be indexed in the same way as an array. Indexers are
20 declared using *indexer-declarations*:

21 *indexer-declaration*:

22 *attributes*_{opt} *indexer-modifiers*_{opt} *indexer-declarator* { *accessor-declarations* }

23 *indexer-modifiers*:

24 *indexer-modifier*

25 *indexer-modifiers* *indexer-modifier*

26 *indexer-modifier*:

27 **new**

28 **public**

29 **protected**

30 **internal**

31 **private**

32 **virtual**

33 **sealed**

34 **override**

35 **abstract**

36 **extern**

37 *indexer-declarator*:

38 *type* **this** [*formal-parameter-list*]

39 *type* *interface-type* . **this** [*formal-parameter-list*]

40 An *indexer-declaration* may include a set of *attributes* (§24) and a valid combination of the four access
41 modifiers (§17.2.3), the **new** (§17.2.2), **virtual** (§17.5.3), **override** (§17.5.4), **sealed** (§17.5.5),
42 **abstract** (§17.5.6), and **extern** (§17.5.7) modifiers.

43 Indexer declarations are subject to the same rules as method declarations (§17.5) with regard to valid
44 combinations of modifiers, with the one exception being that the static modifier is not permitted on an
45 indexer declaration.

1 The modifiers `virtual`, `override`, and `abstract` are mutually exclusive except in one case. The
 2 `abstract` and `override` modifiers may be used together so that an abstract indexer can override a virtual
 3 one.

4 The *type* of an indexer declaration specifies the element type of the indexer introduced by the declaration.
 5 Unless the indexer is an explicit interface member implementation, the *type* is followed by the keyword
 6 `this`. For an explicit interface member implementation, the *type* is followed by an *interface-type*, a `.”`, and
 7 the keyword `this`. Unlike other members, indexers do not have user-defined names.

8 The *formal-parameter-list* specifies the parameters of the indexer. The formal parameter list of an indexer
 9 corresponds to that of a method (§17.5.1), except that at least one parameter must be specified, and that the
 10 `ref` and `out` parameter modifiers are not permitted.

11 The *type* of an indexer and each of the types referenced in the *formal-parameter-list* must be at least as
 12 accessible as the indexer itself (§10.5.4).

13 The *accessor-declarations* (§17.6.2), which must be enclosed in `{` and `}` tokens, declare the accessors
 14 of the indexer. The accessors specify the executable statements associated with reading and writing indexer
 15 elements.

16 Even though the syntax for accessing an indexer element is the same as that for an array element, an indexer
 17 element is not classified as a variable. Thus, it is not possible to pass an indexer element as a `ref` or `out`
 18 argument.

19 The *formal-parameter-list* of an indexer defines the signature (§10.6) of the indexer. Specifically, the
 20 signature of an indexer consists of the number and types of its formal parameters. The element type and
 21 names of the formal parameters are not part of an indexer’s signature.

22 The signature of an indexer must differ from the signatures of all other indexers declared in the same class.

23 Indexers and properties are very similar in concept, but differ in the following ways:

- 24 • A property is identified by its name, whereas an indexer is identified by its signature.
- 25 • A property is accessed through a *simple-name* (§14.5.2) or a *member-access* (§14.5.4), whereas an
 26 indexer element is accessed through an *element-access* (§14.5.6.2).
- 27 • A property can be a `static` member, whereas an indexer is always an instance member.
- 28 • A `get` accessor of a property corresponds to a method with no parameters, whereas a `get` accessor of an
 29 indexer corresponds to a method with the same formal parameter list as the indexer.
- 30 • A `set` accessor of a property corresponds to a method with a single parameter named `value`, whereas a
 31 `set` accessor of an indexer corresponds to a method with the same formal parameter list as the indexer,
 32 plus an additional parameter named `value`.
- 33 • It is a compile-time error for an indexer accessor to declare a local variable with the same name as an
 34 indexer parameter.
- 35 • In an overriding property declaration, the inherited property is accessed using the syntax `base.P`, where
 36 `P` is the property name. In an overriding indexer declaration, the inherited indexer is accessed using the
 37 syntax `base[E]`, where `E` is a comma-separated list of expressions.

38 Aside from these differences, all rules defined in §17.6.2 and §17.6.3 apply to indexer accessors as well as to
 39 property accessors.

40 When an indexer declaration includes an `extern` modifier, the indexer is said to be an *external indexer*.
 41 Because an external indexer declaration provides no actual implementation, each of its *accessor-*
 42 *declarations* consists of a semicolon.

43 [*Example:* The example below declares a `BitArray` class that implements an indexer for accessing the
 44 individual bits in the bit array.

C# LANGUAGE SPECIFICATION

```
1      using System;
2      class BitArray
3      {
4          int[] bits;
5          int length;
6
7          public BitArray(int length) {
8              if (length < 0) throw new ArgumentException();
9              bits = new int[((length - 1) >> 5) + 1];
10             this.length = length;
11         }
12
13         public int Length {
14             get { return length; }
15         }
16
17         public bool this[int index] {
18             get {
19                 if (index < 0 || index >= length) {
20                     throw new IndexOutOfRangeException();
21                 }
22                 return (bits[index >> 5] & 1 << index) != 0;
23             }
24             set {
25                 if (index < 0 || index >= length) {
26                     throw new IndexOutOfRangeException();
27                 }
28                 if (value) {
29                     bits[index >> 5] |= 1 << index;
30                 }
31                 else {
32                     bits[index >> 5] &= ~(1 << index);
33                 }
34             }
35         }
36     }
37 }
```

34 An instance of the `BitArray` class consumes substantially less memory than a corresponding `bool[]`
35 (since each value of the former occupies only one bit instead of the latter's one byte), but it permits the same
36 operations as a `bool[]`.

37 The following `CountPrimes` class uses a `BitArray` and the classical "sieve" algorithm to compute the
38 number of primes between 1 and a given maximum:

```
39      class CountPrimes
40      {
41          static int Count(int max) {
42              BitArray flags = new BitArray(max + 1);
43              int count = 1;
44              for (int i = 2; i <= max; i++) {
45                  if (!flags[i]) {
46                      for (int j = i * 2; j <= max; j += i) flags[j] = true;
47                      count++;
48                  }
49              }
50              return count;
51          }
52
53          static void Main(string[] args) {
54              int max = int.Parse(args[0]);
55              int count = Count(max);
56              Console.WriteLine("Found {0} primes between 1 and {1}", count,
57                               max);
58          }
59      }
```

59 Note that the syntax for accessing elements of the `BitArray` is precisely the same as for a `bool[]`. *end*
60 *example*

[*Example:* The following example shows a 26×10 grid class that has an indexer with two parameters. The first parameter is required to be an upper- or lowercase letter in the range A–Z, and the second is required to be an integer in the range 0–9.

```

4      using System;
5      class Grid
6      {
7          const int NumRows = 26;
8          const int NumCols = 10;
9          int[,] cells = new int[NumRows, NumCols];

10
11         public int this[char c, int colm]
12         {
13             get {
14                 c = Char.ToUpper(c);
15                 if (c < 'A' || c > 'Z') {
16                     throw new ArgumentException();
17                 }
18                 if (colm < 0 || colm >= NumCols) {
19                     throw new IndexOutOfRangeException();
20                 }
21                 return cells[c - 'A', colm];
22             }
23             set {
24                 c = Char.ToUpper(c);
25                 if (c < 'A' || c > 'Z') {
26                     throw new ArgumentException();
27                 }
28                 if (colm < 0 || colm >= NumCols) {
29                     throw new IndexOutOfRangeException();
30                 }
31                 cells[c - 'A', colm] = value;
32             }
33         }
34     }

```

end example]

17.8.1 Indexer overloading

The indexer overload resolution rules are described in §14.4.2.

17.9 Operators

An *operator* is a member that defines the meaning of an expression operator that can be applied to instances of the class. Operators are declared using *operator-declarations*:

```

41     operator-declaration:
42         attributesopt operator-modifiers operator-declarator operator-body
43
44     operator-modifiers:
45         operator-modifier
46         operator-modifiers operator-modifier
47
48     operator-modifier:
49         public
50         static
51         extern
52
53     operator-declarator:
54         unary-operator-declarator
55         binary-operator-declarator
56         conversion-operator-declarator

```

C# LANGUAGE SPECIFICATION

```
1      unary-operator-declarator:  
2      type operator overloadable-unary-operator ( type identifier )  
3  
4      overloadable-unary-operator: one of  
5      + - ! ~ ++ -- true false  
6  
7      binary-operator-declarator:  
8      type operator overloadable-binary-operator ( type identifier , type identifier )  
9  
10     overloadable-binary-operator: one of  
11     + - * / % & | ^ << >> == != > < >= <=  
12  
13     conversion-operator-declarator:  
14     implicit operator type ( type identifier )  
15     explicit operator type ( type identifier )  
16  
17     operator-body:  
18     block  
19     ;
```

There are three categories of overloadable operators: Unary operators (§17.9.1), binary operators (§17.9.2), and conversion operators (§17.9.3).

When an operator declaration includes an `extern` modifier, the operator is said to be an *external operator*. Because an external operator provides no actual implementation, its *operator-body* consists of a semi-colon. For all other operators, the *operator-body* consists of a *block*, which specifies the statements to execute when the operator is invoked. The *block* of an operator must conform to the rules for value-returning methods described in §17.5.8.

The following rules apply to all operator declarations:

- An operator declaration must include both a `public` and a `static` modifier.
- The parameter(s) of an operator must be value parameters. It is a compile-time error for an operator declaration to specify `ref` or `out` parameters.
- The signature of an operator (§17.9.1, §17.9.2, §17.9.3) must differ from the signatures of all other operators declared in the same class.
- All types referenced in an operator declaration must be at least as accessible as the operator itself (§10.5.4).
- It is an error for the same modifier to appear multiple times in an operator declaration.

Each operator category imposes additional restrictions, as described in the following sections.

Like other members, operators declared in a base class are inherited by derived classes. Because operator declarations always require the class or struct in which the operator is declared to participate in the signature of the operator, it is not possible for an operator declared in a derived class to hide an operator declared in a base class. Thus, the `new` modifier is never required, and therefore never permitted, in an operator declaration.

Additional information on unary and binary operators can be found in §14.2.

Additional information on conversion operators can be found in §13.4.

17.9.1 Unary operators

The following rules apply to unary operator declarations, where T denotes the class or struct type that contains the operator declaration:

- A unary `+`, `-`, `!`, or `~` operator must take a single parameter of type T and can return any type.
- A unary `++` or `--` operator must take a single parameter of type T and must return type T.

- A unary `true` or `false` operator must take a single parameter of type `T` and must return type `bool`.

The signature of a unary operator consists of the operator token (`+`, `-`, `!`, `~`, `++`, `--`, `true`, or `false`) and the type of the single formal parameter. The return type is not part of a unary operator's signature, nor is the name of the formal parameter.

The `true` and `false` unary operators require pair-wise declaration. A compile-time error occurs if a class declares one of these operators without also declaring the other. The `true` and `false` operators are described further in §14.16.

[*Example:* The following example shows an implementation and subsequent usage of `operator++` for an integer vector class:

```

10     public class IntVector
11     {
12         public int Length { ... } // read-only property
13         public int this[int index] { ... } // read-write indexer
14         public IntVector(int vectorLength) { ... }
15         public static IntVector operator++(IntVector iv) {
16             IntVector temp = new IntVector(iv.Length);
17             for (int i = 0; i < iv.Length; ++i)
18                 temp[i] = iv[i] + 1;
19             return temp;
20         }
21     }
22     class Test
23     {
24         static void Main() {
25             IntVector iv1 = new IntVector(4); // vector of 4x0
26             IntVector iv2;
27
28             iv2 = iv1++; // iv2 contains 4x0, iv1 contains 4x1
29             iv2 = ++iv1; // iv2 contains 4x2, iv1 contains 4x2
30         }
31     }

```

Note how the operator method returns the value produced by adding 1 to the operand, just like the postfix increment and decrement operators (§14.5.9), and the prefix increment and decrement operators (§14.6.5). Unlike in C++, this method need not, and, in fact, must not, modify the value of its operand directly. *end example*]

17.9.2 Binary operators

A binary operator must take two parameters, at least one of which must have the class or struct type in which the operator is declared. A binary operator can return any type.

The signature of a binary operator consists of the operator token (`+`, `-`, `*`, `/`, `%`, `&`, `|`, `^`, `<<`, `>>`, `==`, `!=`, `>`, `<`, `>=`, or `<=`) and the types of the two formal parameters. The return type and the names of the formal parameters are not part of a binary operator's signature.

Certain binary operators require pair-wise declaration. For every declaration of either operator of a pair, there must be a matching declaration of the other operator of the pair. Two operator declarations match when they have the same return type and the same type for each parameter. The following operators require pair-wise declaration:

- `operator ==` and `operator !=`
- `operator >` and `operator <`
- `operator >=` and `operator <=`

17.9.3 Conversion operators

A conversion operator declaration introduces a *user-defined conversion* (§13.4), which augments the pre-defined implicit and explicit conversions.

C# LANGUAGE SPECIFICATION

1 A conversion operator declaration that includes the `implicit` keyword introduces a user-defined implicit
2 conversion. Implicit conversions can occur in a variety of situations, including function member invocations,
3 cast expressions, and assignments. This is described further in §13.1.

4 A conversion operator declaration that includes the `explicit` keyword introduces a user-defined explicit
5 conversion. Explicit conversions can occur in cast expressions, and are described further in §13.2.

6 A conversion operator converts from a source type, indicated by the parameter type of the conversion
7 operator, to a target type, indicated by the return type of the conversion operator. A class or struct is
8 permitted to declare a conversion from a source type `S` to a target type `T` provided all of the following are
9 true:

- 10 • `S` and `T` are different types.
- 11 • Either `S` or `T` is the class or struct type in which the operator declaration takes place.
- 12 • Neither `S` nor `T` is `object` or an *interface-type*.
- 13 • `T` is not a base class of `S`, and `S` is not a base class of `T`.

14 From the second rule it follows that a conversion operator must convert either to or from the class or struct
15 type in which the operator is declared. [*Example:* For example, it is possible for a class or struct type `C` to
16 define a conversion from `C` to `int` and from `int` to `C`, but not from `int` to `bool`. *end example*]

17 It is not possible to redefine a pre-defined conversion. Thus, conversion operators are not allowed to convert
18 from or to `object` because implicit and explicit conversions already exist between `object` and all other
19 types. Likewise, neither the source nor the target types of a conversion can be a base type of the other, since
20 a conversion would then already exist.

21 User-defined conversions are not allowed to convert from or to *interface-types*. In particular, this restriction
22 ensures that no user-defined transformations occur when converting to an *interface-type*, and that a
23 conversion to an *interface-type* succeeds only if the object being converted actually implements the specified
24 *interface-type*.

25 The signature of a conversion operator consists of the source type and the target type. (Note that this is the
26 only form of member for which the return type participates in the signature.) The `implicit` or `explicit`
27 classification of a conversion operator is not part of the operator's signature. Thus, a class or struct cannot
28 declare both an `implicit` and an `explicit` conversion operator with the same source and target types.

29 [*Note:* In general, user-defined implicit conversions should be designed to never throw exceptions and never
30 lose information. If a user-defined conversion can give rise to exceptions (for example, because the source
31 argument is out of range) or loss of information (such as discarding high-order bits), then that conversion
32 should be defined as an explicit conversion. *end note*]

33 [*Example:* In the example

```
34     using System;
35     public struct Digit
36     {
37         byte value;
38
39         public Digit(byte value) {
40             if (value < 0 || value > 9) throw new ArgumentException();
41             this.value = value;
42         }
43
44         public static implicit operator byte(Digit d) {
45             return d.value;
46         }
47
48         public static explicit operator Digit(byte b) {
49             return new Digit(b);
50         }
51     }
```


1 the conversion from `Digit` to `byte` is implicit because it never throws exceptions or loses information, but
 2 the conversion from `byte` to `Digit` is explicit since `Digit` can only represent a subset of the possible
 3 values of a `byte`. *end example*]

4 **17.10 Instance constructors**

5 An *instance constructor* is a member that implements the actions required to initialize an instance of a class.
 6 Instance constructors are declared using *constructor-declarations*:

```
7     constructor-declaration:
8         attributesopt constructor-modifiersopt constructor-declarator constructor-body
9
10        constructor-modifiers:
11            constructor-modifier
12            constructor-modifiers constructor-modifier
13
14        constructor-modifier:
15            public
16            protected
17            internal
18            private
19            extern
20
21        constructor-declarator:
22            identifier ( formal-parameter-listopt ) constructor-initializeropt
23
24        constructor-initializer:
25            : base ( argument-listopt )
26            : this ( argument-listopt )
27
28        constructor-body:
29            block
30            ;
```

26 A *constructor-declaration* may include a set of *attributes* (§24), a valid combination of the four access
 27 modifiers (§17.2.3), and an `extern` (§17.5.7) modifier. A constructor declaration is not permitted to include
 28 the same modifier multiple times.

29 The *identifier* of a *constructor-declarator* must name the class in which the instance constructor is declared.
 30 If any other name is specified, a compile-time error occurs.

31 The optional *formal-parameter-list* of an instance constructor is subject to the same rules as the *formal-*
 32 *parameter-list* of a method (§17.5). The formal parameter list defines the signature (§10.6) of an instance
 33 constructor and governs the process whereby overload resolution (§14.4.2) selects a particular instance
 34 constructor in an invocation.

35 Each of the types referenced in the *formal-parameter-list* of an instance constructor must be at least as
 36 accessible as the constructor itself (§10.5.4).

37 The optional *constructor-initializer* specifies another instance constructor to invoke before executing the
 38 statements given in the *constructor-body* of this instance constructor. This is described further in §17.10.1.

39 When a constructor declaration includes an `extern` modifier, the constructor is said to be an *external*
 40 *constructor*.

41 Because an external constructor declaration provides no actual implementation, its *constructor-body* consists
 42 of a semicolon. For all other constructors, the *constructor-body* consists of a *block*, which specifies the
 43 statements to initialize a new instance of the class. This corresponds exactly to the *block* of an instance
 44 method with a `void` return type (§17.5.8).

45 Instance constructors are not inherited. Thus, a class has no instance constructors other than those actually
 46 declared in the class. If a class contains no instance constructor declarations, a default instance constructor is
 47 automatically provided (§17.10.4).

1 Instance constructors are invoked by *object-creation-expressions* (§14.5.10.1) and through *constructor-*
2 *initializers*.

3 **17.10.1 Constructor initializers**

4 All instance constructors (except those for class `object`) implicitly include an invocation of another
5 instance constructor immediately before the *constructor-body*. The constructor to implicitly invoke is
6 determined by the *constructor-initializer*:

- 7 • An instance constructor initializer of the form `base(argument-listopt)` causes an instance constructor
8 from the direct base class to be invoked. That constructor is selected using *argument-list* and the
9 overload resolution rules of §14.4.2. The set of candidate instance constructors consists of all accessible
10 instance constructors declared in the direct base class. If this set is empty, or if a single best instance
11 constructor cannot be identified, a compile-time error occurs.
- 12 • An instance constructor initializer of the form `this(argument-listopt)` causes an instance constructor
13 from the class itself to be invoked. The constructor is selected using *argument-list* and the overload
14 resolution rules of §14.4.2. The set of candidate instance constructors consists of all accessible instance
15 constructors declared in the class itself. If that set is empty, or if a single best instance constructor cannot
16 be identified, a compile-time error occurs. If an instance constructor declaration includes a constructor
17 initializer that invokes the constructor itself, a compile-time error occurs.

18 If an instance constructor has no constructor initializer, a constructor initializer of the form `base()` is
19 implicitly provided. [*Note*: Thus, an instance constructor declaration of the form

```
20     C(...) {...}
```

21 is exactly equivalent to

```
22     C(...): base() {...}
```

23 *end note*]

24 The scope of the parameters given by the *formal-parameter-list* of an instance constructor declaration
25 includes the constructor initializer of that declaration. Thus, a constructor initializer is permitted to access
26 the parameters of the constructor. [*Example*: For example:

```
27     class A
28     {
29         public A(int x, int y) {}
30     }
31     class B: A
32     {
33         public B(int x, int y): base(x + y, x - y) {}
34     }
```

35 *end example*]

36 An instance constructor initializer cannot access the instance being created. Therefore it is a compile-time
37 error to reference `this` in an argument expression of the constructor initializer, as it is a compile-time error
38 for an argument expression to reference any instance member through a *simple-name*.

39 **17.10.2 Instance variable initializers**

40 When an instance constructor has no constructor initializer, or it has a constructor initializer of the form
41 `base(...)`, that constructor implicitly performs the initializations specified by the *variable-initializers* of the
42 instance fields declared in its class. This corresponds to a sequence of assignments that are executed
43 immediately upon entry to the constructor and before the implicit invocation of the direct base class
44 constructor. The variable initializers are executed in the textual order in which they appear in the class
45 declaration.

1 17.10.3 Constructor execution

2 Variable initializers are transformed into assignment statements, and these assignment statements are
 3 executed *before* the invocation of the base class instance constructor. This ordering ensures that all instance
 4 fields are initialized by their variable initializers before *any* statements that have access to that instance are
 5 executed. [Example: For example:

```
6     using System;
7     class A
8     {
9         public A() {
10            PrintFields();
11        }
12        public virtual void PrintFields() {}
13    }
14    class B: A
15    {
16        int x = 1;
17        int y;
18        public B() {
19            y = -1;
20        }
21        public override void PrintFields() {
22            Console.WriteLine("x = {0}, y = {1}", x, y);
23        }
24    }
```

25 When `new B()` is used to create an instance of B, the following output is produced:

```
26     x = 1, y = 0
```

27 The value of `x` is 1 because the variable initializer is executed before the base class instance constructor is
 28 invoked. However, the value of `y` is 0 (the default value of an `int`) because the assignment to `y` is not
 29 executed until after the base class constructor returns.

30 It is useful to think of instance variable initializers and constructor initializers as statements that are
 31 automatically inserted before the *constructor-body*. [Example: The example

```
32     using System;
33     using System.Collections;
34     class A
35     {
36         int x = 1, y = -1, count;
37         public A() {
38             count = 0;
39         }
40         public A(int n) {
41             count = n;
42         }
43     }
44     class B: A
45     {
46         double sqrt2 = Math.Sqrt(2.0);
47         ArrayList items = new ArrayList(100);
48         int max;
49         public B(): this(100) {
50             items.Add("default");
51         }
52         public B(int n): base(n - 1) {
53             max = n;
54         }
55     }
```

1 contains several variable initializers; it also contains constructor initializers of both forms (**base** and **this**).
 2 The example corresponds to the code shown below, where each comment indicates an automatically inserted
 3 statement (the syntax used for the automatically inserted constructor invocations isn't valid, but merely
 4 serves to illustrate the mechanism).

```

5     using System.Collections;
6     class A
7     {
8         int x, y, count;
9         public A() {
10            x = 1;           // variable initializer
11            y = -1;        // variable initializer
12            object();      // Invoke object() constructor
13            count = 0;
14        }
15        public A(int n) {
16            x = 1;           // variable initializer
17            y = -1;        // variable initializer
18            object();      // Invoke object() constructor
19            count = n;
20        }
21    }
22    class B: A
23    {
24        double sqrt2;
25        ArrayList items;
26        int max;
27        public B(): this(100) {
28            B(100);         // Invoke B(int) constructor
29            items.Add("default");
30        }
31        public B(int n): base(n - 1) {
32            sqrt2 = Math.Sqrt(2.0); // variable initializer
33            items = new ArrayList(100); // variable initializer
34            A(n - 1);         // Invoke A(int) constructor
35            max = n;
36        }
37    }

```

38 *end example*]

39 17.10.4 Default constructors

40 If a class contains no instance constructor declarations, a default instance constructor is automatically
 41 provided. That default constructor simply invokes the parameterless constructor of the direct base class. If
 42 the direct base class does not have an accessible parameterless instance constructor, a compile-time error
 43 occurs. If the class is abstract then the declared accessibility for the default constructor is protected.
 44 Otherwise, the declared accessibility for the default constructor is public. [*Note*: Thus, the default
 45 constructor is always of the form

```
46     protected C(): base() {}
```

47 or

```
48     public C(): base() {}
```

49 where C is the name of the class. *end note*]

50 [*Example*: In the example

```

51     class Message
52     {
53         object sender;
54         string text;
55     }

```

1 a default constructor is provided because the class contains no instance constructor declarations. Thus, the
2 example is precisely equivalent to

```
3     class Message
4     {
5         object sender;
6         string text;
7
8         public Message(): base() {}
9     }
```

9 *end example]*

10 17.10.5 Private constructors

11 When a class declares only private instance constructors, it is not possible for other classes to derive from
12 that class or to create instances of that class (an exception being classes nested within that class). [*Example:*
13 Private instance constructors are commonly used in classes that contain only static members. For example:

```
14     public class Trig
15     {
16         private Trig() {} // Prevent instantiation
17         public const double PI = 3.14159265358979323846;
18
19         public static double Sin(double x) {...}
20         public static double Cos(double x) {...}
21         public static double Tan(double x) {...}
22     }
```

22 The `Trig` class groups related methods and constants, but is not intended to be instantiated. Therefore, it
23 declares a single empty private instance constructor. *end example]* At least one instance constructor must be
24 declared to suppress the automatic generation of a default constructor.

25 17.10.6 Optional instance constructor parameters

26 [*Note:* The `this(...)` form of constructor initializer is commonly used in conjunction with overloading to
27 implement optional instance constructor parameters. In the example

```
28     class Text
29     {
30         public Text(): this(0, 0, null) {}
31         public Text(int x, int y): this(x, y, null) {}
32         public Text(int x, int y, string s) {
33             // Actual constructor implementation
34         }
35     }
```

36 the first two instance constructors merely provide the default values for the missing arguments. Both use a
37 `this(...)` constructor initializer to invoke the third instance constructor, which actually does the work of
38 initializing the new instance. The effect is that of optional constructor parameters:

```
39     Text t1 = new Text(); // Same as Text(0, 0, null)
40     Text t2 = new Text(5, 10); // Same as Text(5, 10, null)
41     Text t3 = new Text(5, 20, "Hello");
```

42 *end note]*

43 17.11 Static constructors

44 A *static constructor* is a member that implements the actions required to initialize a class. Static
45 constructors are declared using *static-constructor-declarations*:

```
46     static-constructor-declaration:  
47     attributesopt static-constructor-modifiers identifier ( ) static-constructor-body
```

C# LANGUAGE SPECIFICATION

```
1      static-constructor-modifiers:
2          externopt static
3          static externopt
4
5      static-constructor-body:
6          block
7          ;
```

7 A *static-constructor-declaration* may include a set of *attributes* (§24) and an **extern** modifier (§17.5.7).

8 The *identifier* of a *static-constructor-declaration* must name the class in which the static constructor is declared. If any other name is specified, a compile-time error occurs.

10 When a static constructor declaration includes an **extern** modifier, the static constructor is said to be an **external static constructor**. Because an external static constructor declaration provides no actual implementation, its *static-constructor-body* consists of a semicolon. For all other static constructor declarations, the *static-constructor-body* consists of a *block*, which specifies the statements to execute in order to initialize the class. This corresponds exactly to the *method-body* of a static method with a **void** return type (§17.5.8).

16 Static constructors are not inherited, and cannot be called directly.

17 The static constructor for a class executes at most once in a given application domain. The execution of a static constructor is triggered by the first of the following events to occur within an application domain:

- 19 • An instance of the class is created.
- 20 • Any of the static members of the class are referenced.

21 If a class contains the **Main** method (§10.1) in which execution begins, the static constructor for that class executes before the **Main** method is called. If a class contains any static fields with initializers, those initializers are executed in textual order immediately prior to executing the static constructor.

24 [*Example*: The example

```
25     using System;
26     class Test
27     {
28         static void Main() {
29             A.F();
30             B.F();
31         }
32     }
33     class A
34     {
35         static A() {
36             Console.WriteLine("Init A");
37         }
38         public static void F() {
39             Console.WriteLine("A.F");
40         }
41     }
42     class B
43     {
44         static B() {
45             Console.WriteLine("Init B");
46         }
47         public static void F() {
48             Console.WriteLine("B.F");
49         }
50     }
```

51 must produce the output:

```

1      Init A
2      A.F
3      Init B
4      B.F

```

5 because the execution of A's static constructor is triggered by the call to A.F, and the execution of B's static
6 constructor is triggered by the call to B.F. *end example*]

7 It is possible to construct circular dependencies that allow static fields with variable initializers to be
8 observed in their default value state.

9 [*Example*: The example

```

10     using System;
11     class A
12     {
13         public static int X;
14         static A() { X = B.Y + 1;}
15     }
16     class B
17     {
18         public static int Y = A.X + 1;
19         static B() {}
20         static void Main() {
21             Console.WriteLine("X = {0}, Y = {1}", A.X, B.Y);
22         }
23     }

```

24 produces the output

```

25     X = 1, Y = 2

```

26 To execute the Main method, the system first runs the initializer for B.Y, prior to class B's static constructor.
27 Y's initializer causes A's static constructor to be run because the value of A.X is referenced. The static
28 constructor of A in turn proceeds to compute the value of X, and in doing so fetches the default value of Y,
29 which is zero. A.X is thus initialized to 1. The process of running A's static field initializers and static
30 constructor then completes, returning to the calculation of the initial value of Y, the result of which
31 becomes 2. *end example*]

32 17.12 Destructors

33 A **destructor** is a member that implements the actions required to destruct an instance of a class. A
34 destructor is declared using a *destructor-declaration*:

```

35     destructor-declaration:
36         attributesopt externopt ~ identifier ( ) destructor-body
37
38     destructor-body:
39         block
40         ;

```

40 A *destructor-declaration* may include a set of *attributes* (§24).

41 The *identifier* of a *destructor-declaration* must name the class in which the destructor is declared. If any other
42 name is specified, a compile-time error occurs.

43 When a destructor declaration includes an **extern** modifier, the destructor is said to be an **external**
44 **destructor**. Because an external destructor declaration provides no actual implementation, its *destructor-*
45 *body* consists of a semicolon. For all other destructors, the *destructor-body* consists of a *block*, which
46 specifies the statements to execute in order to destruct an instance of the class. A *destructor-body*
47 corresponds exactly to the *method-body* of an instance method with a void return type (§17.5.8).

48 Destructors are not inherited. Thus, a class has no destructors other than the one which may be declared in
49 that class.

C# LANGUAGE SPECIFICATION

1 [Note: Since a destructor is required to have no parameters, it cannot be overloaded, so a class can have, at
2 most, one destructor. *end note*]

3 Destructors are invoked automatically, and cannot be invoked explicitly. An instance becomes eligible for
4 destruction when it is no longer possible for any code to use that instance. Execution of the destructor for the
5 instance may occur at any time after the instance becomes eligible for destruction. When an instance is
6 destructed, the destructors in that instance's inheritance chain are called, in order, from most derived to least
7 derived [*Example*: The output of the example]

```
8     using System;
9     class A
10    {
11        ~A() {
12            Console.WriteLine("A's destructor");
13        }
14    }
15    class B: A
16    {
17        ~B() {
18            Console.WriteLine("B's destructor");
19        }
20    }
21    class Test
22    {
23        static void Main() {
24            B b = new B();
25            b = null;
26            GC.Collect();
27            GC.WaitForPendingFinalizers();
28        }
29    }
```

30 is

```
31     B's destructor
32     A's destructor
```

33 since destructors in an inheritance chain are called in order, from most derived to least derived. *end example*]

34 Destructors may be implemented by overriding the virtual method `Finalize` on `System.Object`. In any
35 event, C# programs are not permitted to override this method or call it (or overrides of it) directly.

36 [*Example*: For instance, the program

```
37     class A
38     {
39         override protected void Finalize() {} // error
40         public void F() {
41             this.Finalize(); // error
42         }
43     }
```

44 contains two errors. *end example*]

45 The compiler behaves as if this method, and overrides of it, does not exist at all. [*Example*: Thus, this
46 program:

```
47     class A
48     {
49         void Finalize() {} // permitted
50     }
```

51 is valid and the method shown hides `System.Object`'s `Finalize` method. *end example*]

52 For a discussion of the behavior when an exception is thrown from a destructor, see §23.3.

18. Structs

1

2 Structs are similar to classes in that they represent data structures that can contain data members and
 3 function members. However, unlike classes, structs are value types and do not require heap allocation. A
 4 variable of a struct type directly contains the data of the struct, whereas a variable of a class type contains a
 5 reference to the data, the latter known as an object.

6 [*Note:* Structs are particularly useful for small data structures that have value semantics. Complex numbers,
 7 points in a coordinate system, or key-value pairs in a dictionary are all good examples of structs. Key to
 8 these data structures is that they have few data members, that they do not require use of inheritance or
 9 referential identity, and that they can be conveniently implemented using value semantics where assignment
 10 copies the value instead of the reference. *end note*]

11 As described in §11.1.3, the simple types provided by C#, such as `int`, `double`, and `bool`, are, in fact, all
 12 struct types. Just as these predefined types are structs, it is also possible to use structs and operator
 13 overloading to implement new “primitive” types in the C# language. Two examples of such types are given
 14 at the end of this chapter (§18.4).

15 18.1 Struct declarations

16 A *struct-declaration* is a *type-declaration* (§16.5) that declares a new struct:

17 *struct-declaration:*
 18 *attributes*_{opt} *struct-modifiers*_{opt} **struct** *identifier* *struct-interfaces*_{opt} *struct-body* ;_{opt}

19 A *struct-declaration* consists of an optional set of *attributes* (§24), followed by an optional set of *struct-*
 20 *modifiers* (§18.1.1), followed by the keyword **struct** and an *identifier* that names the struct, followed by an
 21 optional *struct-interfaces* specification (§18.1.2), followed by a *struct-body* (§18.1.3), optionally followed
 22 by a semicolon.

23 18.1.1 Struct modifiers

24 A *struct-declaration* may optionally include a sequence of struct modifiers:

25 *struct-modifiers:*
 26 *struct-modifier*
 27 *struct-modifiers* *struct-modifier*

28 *struct-modifier:*
 29 **new**
 30 **public**
 31 **protected**
 32 **internal**
 33 **private**

34 It is a compile-time error for the same modifier to appear multiple times in a struct declaration.

35 The modifiers of a struct declaration have the same meaning as those of a class declaration (§17.1.1).

36 18.1.2 Struct interfaces

37 A struct declaration may include a *struct-interfaces* specification, in which case the struct is said to
 38 implement the given interface types.

39 *struct-interfaces:*
 40 **:** *interface-type-list*

1 Interface implementations are discussed further in §20.4.

2 **18.1.3 Struct body**

3 The *struct-body* of a struct defines the members of the struct.

```
4     struct-body:
5         { struct-member-declarationsopt }
```

6 **18.2 Struct members**

7 The members of a struct consist of the members introduced by its *struct-member-declarations* and the
8 members inherited from the type `System.ValueType`.

```
9     struct-member-declarations:
10         struct-member-declaration
11         struct-member-declarations struct-member-declaration
12
13     struct-member-declaration:
14         constant-declaration
15         field-declaration
16         method-declaration
17         property-declaration
18         event-declaration
19         indexer-declaration
20         operator-declaration
21         constructor-declaration
22         static-constructor-declaration
23         type-declaration
```

23 Except for the differences noted in §18.3, the descriptions of class members provided in §17.2 through
24 §17.11 apply to struct members as well.

25 **18.3 Class and struct differences**

26 **18.3.1 Value semantics**

27 Structs are value types (§11.1) and are said to have value semantics. Classes, on the other hand, are reference
28 types (§11.2) and are said to have reference semantics.

29 A variable of a struct type directly contains the data of the struct, whereas a variable of a class type contains
30 a reference to the data, the latter known as an object.

31 With classes, it is possible for two variables to reference the same object, and thus possible for operations on
32 one variable to affect the object referenced by the other variable. With structs, the variables each have their
33 own copy of the data, and it is not possible for operations on one to affect the other. Furthermore, because
34 structs are not reference types, it is not possible for values of a struct type to be `null`.

35 [*Example*: Given the declaration

```
36     struct Point
37     {
38         public int x, y;
39         public Point(int x, int y) {
40             this.x = x;
41             this.y = y;
42         }
43     }
```

44 the code fragment

```

1      Point a = new Point(10, 10);
2      Point b = a;
3      a.x = 100;
4      System.Console.WriteLine(b.x);

```

5 outputs the value 10. The assignment of `a` to `b` creates a copy of the value, and `b` is thus unaffected by the
6 assignment to `a.x`. Had `Point` instead been declared as a class, the output would be 100 because `a` and `b`
7 would reference the same object. *end example*]

8 18.3.2 Inheritance

9 All struct types implicitly inherit from `System.ValueType`, which, in turn, inherits from class `object`. A
10 struct declaration may specify a list of implemented interfaces, but it is not possible for a struct declaration
11 to specify a base class.

12 Struct types are never abstract and are always implicitly sealed. The `abstract` and `sealed` modifiers are
13 therefore not permitted in a struct declaration.

14 Since inheritance isn't supported for structs, the declared accessibility of a struct member cannot be
15 `protected` or `protected internal`.

16 Function members in a struct cannot be `abstract` or `virtual`, and the `override` modifier is allowed
17 only to override methods inherited from the type `System.ValueType`.

18 18.3.3 Assignment

19 Assignment to a variable of a struct type creates a *copy* of the value being assigned. This differs from
20 assignment to a variable of a class type, which copies the reference but not the object identified by the
21 reference.

22 Similar to an assignment, when a struct is passed as a value parameter or returned as the result of a function
23 member, a copy of the struct is created. A struct may be passed by reference to a function member using a
24 `ref` or `out` parameter.

25 When a property or indexer of a struct is the target of an assignment, the instance expression associated with
26 the property or indexer access must be classified as a variable. If the instance expression is classified as a
27 value, a compile-time error occurs. This is described in further detail in §14.13.1.

28 18.3.4 Default values

29 As described in §12.2, several kinds of variables are automatically initialized to their default value when
30 they are created. For variables of class types and other reference types, this default value is `null`. However,
31 since structs are value types that cannot be `null`, the default value of a struct is the value produced by
32 setting all value type fields to their default value and all reference type fields to `null`.

33 [*Example:* Referring to the `Point` struct declared above, the example

```

34      Point[] a = new Point[100];

```

35 initializes each `Point` in the array to the value produced by setting the `x` and `y` fields to zero. *end example*]

36 The default value of a struct corresponds to the value returned by the default constructor of the struct
37 (§11.1.1). Unlike a class, a struct is not permitted to declare a parameterless instance constructor. Instead,
38 every struct implicitly has a parameterless instance constructor, which always returns the value that results
39 from setting all value type fields to their default value and all reference type fields to `null`.

40 [*Note:* Structs should be designed to consider the default initialization state a valid state. In the example

```

41      using System;
42      struct KeyValuePair
43      {
44          string key;
45          string value;

```

```

1     public KeyValuePair(string key, string value) {
2         if (key == null || value == null) throw new ArgumentException();
3         this.key = key;
4         this.value = value;
5     }
6 }

```

7 the user-defined instance constructor protects against null values only where it is explicitly called. In cases
8 where a `KeyValuePair` variable is subject to default value initialization, the `key` and `value` fields will be
9 null, and the struct must be prepared to handle this state. *end note*

10 18.3.5 Boxing and unboxing

11 A value of a class type can be converted to type `object` or to an interface type that is implemented by the
12 class simply by treating the reference as another type at compile-time. Likewise, a value of type `object` or
13 a value of an interface type can be converted back to a class type without changing the reference (but of
14 course a run-time type check is required in this case).

15 Since structs are not reference types, these operations are implemented differently for struct types. When a
16 value of a struct type is converted to type `object` or to an interface type that is implemented by the struct, a
17 boxing operation takes place. Likewise, when a value of type `object` or a value of an interface type is
18 converted back to a struct type, an unboxing operation takes place. A key difference from the same
19 operations on class types is that boxing and unboxing *copies* the struct value either into or out of the boxed
20 instance. [*Note:* Thus, following a boxing or unboxing operation, changes made to the unboxed struct are not
21 reflected in the boxed struct. *end note*]

22 For further details on boxing and unboxing, see §11.3.

23 18.3.6 Meaning of `this`

24 Within an instance constructor or instance function member of a class, `this` is classified as a value. Thus,
25 while `this` can be used to refer to the instance for which the function member was invoked, it is not
26 possible to assign to `this` in a function member of a class.

27 Within an instance constructor of a struct, `this` corresponds to an `out` parameter of the struct type, and
28 within an instance function member of a struct, `this` corresponds to a `ref` parameter of the struct type. In
29 both cases, `this` is classified as a variable, and it is possible to modify the entire struct for which the
30 function member was invoked by assigning to `this` or by passing `this` as a `ref` or `out` parameter.

31 18.3.7 Field initializers

32 As described in §18.3.4, the default value of a struct consists of the value that results from setting all value
33 type fields to their default value and all reference type fields to `null`. For this reason, a struct does not
34 permit instance field declarations to include variable initializers. [*Example:* As such, the following example
35 results in one or more compile-time errors:

```

36     struct Point
37     {
38         public int x = 1; // Error, initializer not permitted
39         public int y = 1; // Error, initializer not permitted
40     }

```

41 *end example*]

42 This restriction applies only to instance fields. Static fields of a struct are permitted to include variable
43 initializers.

44 18.3.8 Constructors

45 Unlike a class, a struct is not permitted to declare a parameterless instance constructor. Instead, every struct
46 implicitly has a parameterless instance constructor, which always returns the value that results from setting
47 all value type fields to their default value and all reference type fields to `null` (§11.1.1). A struct can declare
48 instance constructors having parameters. [*Example:* For example

```

1      struct Point
2      {
3          int x, y;
4          public Point(int x, int y) {
5              this.x = x;
6              this.y = y;
7          }
8      }

```

9 Given the above declaration, the statements

```

10     Point p1 = new Point();
11     Point p2 = new Point(0, 0);

```

12 both create a `Point` with `x` and `y` initialized to zero. *end example*]

13 A struct instance constructor is not permitted to include a constructor initializer of the form `base(...)`.

14 The `this` variable of a struct instance constructor corresponds to an `out` parameter of the struct type, and similar to an `out` parameter, `this` must be definitely assigned (§12.3) at every location where the constructor returns. [*Example*: Consider the instance constructor implementation below:

```

17     struct Point
18     {
19         int x, y;
20         public int X {
21             set { x = value; }
22         }
23         public int Y {
24             set { y = value; }
25         }
26         public Point(int x, int y) {
27             X = x;           // error, this is not yet definitely assigned
28             Y = y;           // error, this is not yet definitely assigned
29         }
30     }

```

31 No instance member function (including the set accessors for the properties `X` and `Y`) can be called until all fields of the struct being constructed have been definitely assigned. Note, however, that if `Point` were a class instead of a struct, the instance constructor implementation would be permitted.

34 *end example*]

35 18.3.9 Destructors

36 A struct is not permitted to declare a destructor.

37 18.4 Struct examples

38 **This whole clause is informative.**

39 18.4.1 Database integer type

40 The `DBInt` struct below implements an integer type that can represent the complete set of values of the `int` type, plus an additional state that indicates an unknown value. A type with these characteristics is commonly used in databases.

```

43     using System;
44     public struct DBInt
45     {
46         // The Null member represents an unknown DBInt value.
47         public static readonly DBInt Null = new DBInt();

```

C# LANGUAGE SPECIFICATION

```
1         // when the defined field is true, this DBInt represents a known value
2         // which is stored in the value field. when the defined field is
3         false,
4         // this DBInt represents an unknown value, and the value field is 0.
5         int value;
6         bool defined;
7         // Private instance constructor. Creates a DBInt with a known value.
8         DBInt(int value) {
9             this.value = value;
10            this.defined = true;
11        }
12        // The IsNull property is true if this DBInt represents an unknown
13        value.
14        public bool IsNull { get { return !defined; } }
15        // The Value property is the known value of this DBInt, or 0 if this
16        // DBInt represents an unknown value.
17        public int value { get { return value; } }
18        // Implicit conversion from int to DBInt.
19        public static implicit operator DBInt(int x) {
20            return new DBInt(x);
21        }
22        // Explicit conversion from DBInt to int. Throws an exception if the
23        // given DBInt represents an unknown value.
24        public static explicit operator int(DBInt x) {
25            if (!x.defined) throw new InvalidOperationException();
26            return x.value;
27        }
28        public static DBInt operator +(DBInt x) {
29            return x;
30        }
31        public static DBInt operator -(DBInt x) {
32            return x.defined? -x.value: Null;
33        }
34        public static DBInt operator +(DBInt x, DBInt y) {
35            return x.defined && y.defined? x.value + y.value: Null;
36        }
37        public static DBInt operator -(DBInt x, DBInt y) {
38            return x.defined && y.defined? x.value - y.value: Null;
39        }
40        public static DBInt operator *(DBInt x, DBInt y) {
41            return x.defined && y.defined? x.value * y.value: Null;
42        }
43        public static DBInt operator /(DBInt x, DBInt y) {
44            return x.defined && y.defined? x.value / y.value: Null;
45        }
46        public static DBInt operator %(DBInt x, DBInt y) {
47            return x.defined && y.defined? x.value % y.value: Null;
48        }
49        public static DBBool operator ==(DBInt x, DBInt y) {
50            return x.defined && y.defined? x.value == y.value: DBBool.Null;
51        }
52        public static DBBool operator !=(DBInt x, DBInt y) {
53            return x.defined && y.defined? x.value != y.value: DBBool.Null;
54        }
55        public static DBBool operator >(DBInt x, DBInt y) {
56            return x.defined && y.defined? x.value > y.value: DBBool.Null;
57        }
```

```

1     public static DBBool operator <(DBInt x, DBInt y) {
2         return x.defined && y.defined? x.value < y.value: DBBool.Null;
3     }
4     public static DBBool operator >=(DBInt x, DBInt y) {
5         return x.defined && y.defined? x.value >= y.value: DBBool.Null;
6     }
7     public static DBBool operator <=(DBInt x, DBInt y) {
8         return x.defined && y.defined? x.value <= y.value: DBBool.Null;
9     }
10    }

```

11 18.4.2 Database boolean type

12 The DBBool struct below implements a three-valued logical type. The possible values of this type are
13 DBBool.True, DBBool.False, and DBBool.Null, where the Null member indicates an unknown value.
14 Such three-valued logical types are commonly used in databases.

```

15     using System;
16     public struct DBBool
17     {
18         // The three possible DBBool values.
19         public static readonly DBBool Null = new DBBool(0);
20         public static readonly DBBool False = new DBBool(-1);
21         public static readonly DBBool True = new DBBool(1);
22         // Private field that stores -1, 0, 1 for False, Null, True.
23         sbyte value;
24         // Private instance constructor. The value parameter must be -1, 0, or
25         1.
26         DBBool(int value) {
27             this.value = (sbyte)value;
28         }
29         // Properties to examine the value of a DBBool. Return true if this
30         // DBBool has the given value, false otherwise.
31         public bool IsNull { get { return value == 0; } }
32         public bool IsFalse { get { return value < 0; } }
33         public bool IsTrue { get { return value > 0; } }
34         // Implicit conversion from bool to DBBool. Maps true to DBBool.True
35         and // false to DBBool.False.
36         public static implicit operator DBBool(bool x) {
37             return x? True: False;
38         }
39         // Explicit conversion from DBBool to bool. Throws an exception if the
40         // given DBBool is Null, otherwise returns true or false.
41         public static explicit operator bool(DBBool x) {
42             if (x.value == 0) throw new InvalidOperationException();
43             return x.value > 0;
44         }
45         // Equality operator. Returns Null if either operand is Null,
46         otherwise // returns True or False.
47         public static DBBool operator ==(DBBool x, DBBool y) {
48             if (x.value == 0 || y.value == 0) return Null;
49             return x.value == y.value? True: False;
50         }
51         // Inequality operator. Returns Null if either operand is Null,
52         otherwise // returns True or False.
53         public static DBBool operator !=(DBBool x, DBBool y) {
54             if (x.value == 0 || y.value == 0) return Null;
55             return x.value != y.value? True: False;
56         }
57     }

```

C# LANGUAGE SPECIFICATION

```
1     public static DBBool operator !=(DBBool x, DBBool y) {
2         if (x.value == 0 || y.value == 0) return Null;
3         return x.value != y.value? True: False;
4     }
5     // Logical negation operator. Returns True if the operand is False,
6     Null
7     // if the operand is Null, or False if the operand is True.
8     public static DBBool operator !(DBBool x) {
9         return new DBBool(-x.value);
10    }
11    // Logical AND operator. Returns False if either operand is False,
12    // otherwise Null if either operand is Null, otherwise True.
13    public static DBBool operator &(amp;DBBool x, DBBool y) {
14        return new DBBool(x.value < y.value? x.value: y.value);
15    }
16    // Logical OR operator. Returns True if either operand is True,
17    otherwise
18    // Null if either operand is Null, otherwise False.
19    public static DBBool operator |(DBBool x, DBBool y) {
20        return new DBBool(x.value > y.value? x.value: y.value);
21    }
22    // Definitely true operator. Returns true if the operand is True,
23    false
24    // otherwise.
25    public static bool operator true(DBBool x) {
26        return x.value > 0;
27    }
28    // Definitely false operator. Returns true if the operand is False,
29    false
30    // otherwise.
31    public static bool operator false(DBBool x) {
32        return x.value < 0;
33    }
34 }
```

35 **End of informative text.**

19. Arrays

1

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

5 An array has a rank which determines the number of indices associated with each array element. The rank of
 6 an array is also referred to as the dimensions of the array. An array with a rank of one is called a *single-*
 7 *dimensional array*. An array with a rank greater than one is called a *multi-dimensional array*. Specific sized
 8 multi-dimensional arrays are often referred to as two-dimensional arrays, three-dimensional arrays, and so
 9 on. Each dimension of an array has an associated length which is an integral number greater than or equal to
 10 zero. The dimension lengths are not part of the type of the array, but rather are established when an instance
 11 of the array type is created at run-time. The length of a dimension determines the valid range of indices for
 12 that dimension: For a dimension of length N , indices can range from 0 to $N - 1$ inclusive. The total number
 13 of elements in an array is the product of the lengths of each dimension in the array. If one or more of the
 14 dimensions of an array have a length of zero, the array is said to be empty.

15 The element type of an array can be any type, including an array type.

16 19.1 Array types

17 An array type is written as a *non-array-type* followed by one or more *rank-specifiers*:

```
18     array-type:
19         non-array-type rank-specifiers
20
21     non-array-type:
22         type
23
24     rank-specifiers:
25         rank-specifier
26         rank-specifiers rank-specifier
27
28     rank-specifier:
29         [ dim-separatorsopt ]
30
31     dim-separators:
32         ,
33         dim-separators ,
```

30 A *non-array-type* is any *type* that is not itself an *array-type*.

31 The rank of an array type is given by the leftmost *rank-specifier* in the *array-type*: A *rank-specifier* indicates
 32 that the array is an array with a rank of one plus the number of “,” tokens in the *rank-specifier*.

33 The element type of an array type is the type that results from deleting the leftmost *rank-specifier*:

- 34 • An array type of the form $T[R]$ is an array with rank R and a non-array element type T .
- 35 • An array type of the form $T[R][R_1] \dots [R_N]$ is an array with rank R and an element type $T[R_1] \dots [R_N]$.

36 In effect, the *rank-specifiers* are read from left to right *before* the final non-array element type. [Example:
 37 The type `int [] [,] [,]` is a single-dimensional array of three-dimensional arrays of two-dimensional
 38 arrays of `int`. *end example*]

39 At run-time, a value of an array type can be `null` or a reference to an instance of that array type.

19.1.1 The System.Array type

The type `System.Array` is the abstract base type of all array types. An implicit reference conversion (§13.1.4) exists from any array type to `System.Array`, and an explicit reference conversion (§13.2.3) exists from `System.Array` to any array type. Note that `System.Array` is not itself an *array-type*. Rather, it is a *class-type* from which all *array-types* are derived.

At run-time, a value of type `System.Array` can be `null` or a reference to an instance of any array type.

19.2 Array creation

Array instances are created by *array-creation-expressions* (§14.5.10.2) or by field or local variable declarations that include an *array-initializer* (§19.6).

When an array instance is created, the rank and length of each dimension are established and then remain constant for the entire lifetime of the instance. In other words, it is not possible to change the rank of an existing array instance, nor is it possible to resize its dimensions.

An array instance is always of an array type. The `System.Array` type is an abstract type that cannot be instantiated.

Elements of arrays created by *array-creation-expressions* are always initialized to their default value (§12.2).

19.3 Array element access

Array elements are accessed using *element-access* expressions (§14.5.6.1) of the form `A[I1, I2, ..., IN]`, where `A` is an expression of an array type and each `Ix` is an expression of type `int`, `uint`, `long`, `ulong`, or of a type that can be implicitly converted to one or more of these types. The result of an array element access is a variable, namely the array element selected by the indices.

The elements of an array can be enumerated using a `foreach` statement (§15.8.4).

19.4 Array members

Every array type inherits the members declared by the `System.Array` type.

19.5 Array covariance

For any two *reference-types* `A` and `B`, if an implicit reference conversion (§13.1.4) or explicit reference conversion (§13.2.3) exists from `A` to `B`, then the same reference conversion also exists from the array type `A[R]` to the array type `B[R]`, where `R` is any given *rank-specifier* (but the same for both array types). This relationship is known as **array covariance**. Array covariance, in particular, means that a value of an array type `A[R]` may actually be a reference to an instance of an array type `B[R]`, provided an implicit reference conversion exists from `B` to `A`.

Because of array covariance, assignments to elements of reference type arrays include a run-time check which ensures that the value being assigned to the array element is actually of a permitted type (§14.13.1).

[*Example:* For example:

```

35     class Test
36     {
37         static void Fill(object[] array, int index, int count, object value) {
38             for (int i = index; i < index + count; i++) array[i] = value;
39         }
40
41         static void Main() {
42             string[] strings = new string[100];
43             Fill(strings, 0, 100, "Undefined");
44             Fill(strings, 0, 10, null);
45             Fill(strings, 90, 10, 0);
46         }

```

1 The assignment to `array[i]` in the `Fill` method implicitly includes a run-time check, which ensures that
 2 the object referenced by `value` is either `null` or an instance of a type that is compatible with the actual
 3 element type of `array`. In `Main`, the first two invocations of `Fill` succeed, but the third invocation causes a
 4 `System.ArrayTypeMismatchException` to be thrown upon executing the first assignment to
 5 `array[i]`. The exception occurs because a boxed `int` cannot be stored in a `string` array. *end example*]

6 Array covariance specifically does not extend to arrays of *value-types*. For example, no conversion exists
 7 that permits an `int[]` to be treated as an `object[]`.

8 19.6 Array initializers

9 Array initializers may be specified in field declarations (§17.4), local variable declarations (§15.5.1), and
 10 array creation expressions (§14.5.10.2):

```
11     array-initializer:
12         { variable-initializer-listopt }
13         { variable-initializer-list , }
14
15     variable-initializer-list:
16         variable-initializer
17         variable-initializer-list , variable-initializer
18
19     variable-initializer:
20         expression
21         array-initializer
```

22 An array initializer consists of a sequence of variable initializers, enclosed by “{” and “}” tokens and
 23 separated by “,” tokens. Each variable initializer is an expression or, in the case of a multi-dimensional
 24 array, a nested array initializer.

25 The context in which an array initializer is used determines the type of the array being initialized. In an array
 26 creation expression, the array type immediately precedes the initializer. In a field or variable declaration, the
 27 array type is the type of the field or variable being declared. When an array initializer is used in a field or
 28 variable declaration, [*Example*: such as:

```
29     int[] a = {0, 2, 4, 6, 8};
```

30 *end example*] it is simply shorthand for an equivalent array creation expression: [*Example*:

```
31     int[] a = new int[] {0, 2, 4, 6, 8};
```

32 *end example*]

33 For a single-dimensional array, the array initializer must consist of a sequence of expressions that are
 34 assignment compatible with the element type of the array. The expressions initialize array elements in
 35 increasing order, starting with the element at index zero. The number of expressions in the array initializer
 36 determines the length of the array instance being created. [*Example*: For example, the array initializer above
 37 creates an `int[]` instance of length 5 and then initializes the instance with the following values:

```
38     a[0] = 0; a[1] = 2; a[2] = 4; a[3] = 6; a[4] = 8;
```

39 *end example*]

40 For a multi-dimensional array, the array initializer must have as many levels of nesting as there are
 41 dimensions in the array. The outermost nesting level corresponds to the leftmost dimension and the
 42 innermost nesting level corresponds to the rightmost dimension. The length of each dimension of the array is
 43 determined by the number of elements at the corresponding nesting level in the array initializer. For each
 44 nested array initializer, the number of elements must be the same as the other array initializers at the same
 45 level. [*Example*: The example:

```
46     int[,] b = {{0, 1}, {2, 3}, {4, 5}, {6, 7}, {8, 9}};
```

creates a two-dimensional array with a length of five for the leftmost dimension and a length of two for the
 rightmost dimension:

C# LANGUAGE SPECIFICATION

1 int[,] b = new int[5, 2];

2 and then initializes the array instance with the following values:

```
3           b[0, 0] = 0; b[0, 1] = 1;
4           b[1, 0] = 2; b[1, 1] = 3;
5           b[2, 0] = 4; b[2, 1] = 5;
6           b[3, 0] = 6; b[3, 1] = 7;
7           b[4, 0] = 8; b[4, 1] = 9;
```

8 *end example]*

9 When an array creation expression includes both explicit dimension lengths and an array initializer, the
10 lengths must be constant expressions and the number of elements at each nesting level must match the
11 corresponding dimension length. [*Example:* Here are some examples:

```
12           int i = 3;
13           int[] x = new int[3] {0, 1, 2};        // OK
14           int[] y = new int[i] {0, 1, 2};       // Error, i not a constant
15           int[] z = new int[3] {0, 1, 2, 3};    // Error, length/initializer mismatch
```

16 Here, the initializer for y results in a compile-time error because the dimension length expression is not a
17 constant, and the initializer for z results in a compile-time error because the length and the number of
18 elements in the initializer do not agree. *end example]*

20. Interfaces

1

2 An interface defines a contract. A class or struct that implements an interface must adhere to its contract. An
3 interface may inherit from multiple base interfaces, and a class or struct may implement multiple interfaces.

4 Interfaces can contain methods, properties, events, and indexers. The interface itself does not provide
5 implementations for the members that it defines. The interface merely specifies the members that must be
6 supplied by classes or interfaces that implement the interface.

7 20.1 Interface declarations

8 An *interface-declaration* is a *type-declaration* (§16.5) that declares a new interface type.

9 *interface-declaration:*
10 *attributes*_{opt} *interface-modifiers*_{opt} **interface** *identifier* *interface-base*_{opt} *interface-body*
11 *;*_{opt}

12 An *interface-declaration* consists of an optional set of *attributes* (§24), followed by an optional set of
13 *interface-modifiers* (§20.1.1), followed by the keyword **interface** and an *identifier* that names the
14 interface, optionally followed by an optional *interface-base* specification (§20.1.2), followed by a *interface-*
15 *body* (§20.1.3), optionally followed by a semicolon.

16 20.1.1 Interface modifiers

17 An *interface-declaration* may optionally include a sequence of interface modifiers:

18 *interface-modifiers:*
19 *interface-modifier*
20 *interface-modifiers* *interface-modifier*

21 *interface-modifier:*
22 **new**
23 **public**
24 **protected**
25 **internal**
26 **private**

27 It is a compile-time error for the same modifier to appear multiple times in an interface declaration.

28 The **new** modifier is only permitted on nested interfaces. It specifies that the interface hides an inherited
29 member by the same name, as described in §17.2.2.

30 The **public**, **protected**, **internal**, and **private** modifiers control the accessibility of the interface.
31 Depending on the context in which the interface declaration occurs, only some of these modifiers may be
32 permitted (§10.5.1).

33 20.1.2 Base interfaces

34 An interface can inherit from zero or more interfaces, which are called the *explicit base interfaces* of the
35 interface. When an interface has one or more explicit base interfaces, then in the declaration of that interface,
36 the interface identifier is followed by a colon and a comma-separated list of base interface identifiers.

37 *interface-base:*
38 *:* *interface-type-list*

1 The explicit base interfaces of an interface must be at least as accessible as the interface itself (§10.5.4).
 2 [Note: For example, it is a compile-time error to specify a `private` or `internal` interface in the *interface-*
 3 *base* of a `public` interface. *end note*]

4 It is a compile-time error for an interface to directly or indirectly inherit from itself.

5 The **base interfaces** of an interface are the explicit base interfaces and their base interfaces. In other words,
 6 the set of base interfaces is the complete transitive closure of the explicit base interfaces, their explicit base
 7 interfaces, and so on. An interface inherits all members of its base interfaces. [Example: In the example

```

  8     interface IControl
  9     {
10         void Paint();
11     }
12     interface ITextBox: IControl
13     {
14         void SetText(string text);
15     }
16     interface IListBox: IControl
17     {
18         void SetItems(string[] items);
19     }
20     interface IComboBox: ITextBox, IListBox {}

```

21 the base interfaces of `IComboBox` are `IControl`, `ITextBox`, and `IListBox`. In other words, the
 22 `IComboBox` interface above inherits members `SetText` and `SetItems` as well as `Paint`. *end example*]

23 A class or struct that implements an interface also implicitly implements all of the interface's base
 24 interfaces.

25 20.1.3 Interface body

26 The *interface-body* of an interface defines the members of the interface.

```

27     interface-body:
28         { interface-member-declarationsopt }

```

29 20.2 Interface members

30 The members of an interface are the members inherited from the base interfaces and the members declared
 31 by the interface itself.

```

32     interface-member-declarations:
33         interface-member-declaration
34         interface-member-declarations interface-member-declaration

```

```

35     interface-member-declaration:
36         interface-method-declaration
37         interface-property-declaration
38         interface-event-declaration
39         interface-indexer-declaration

```

40 An interface declaration may declare zero or more members. The members of an interface must be methods,
 41 properties, events, or indexers. An interface cannot contain constants, fields, operators, instance
 42 constructors, destructors, or types, nor can an interface contain static members of any kind.

43 All interface members implicitly have public access. It is a compile-time error for interface member
 44 declarations to include any modifiers. In particular, interface members cannot be declared with the modifiers
 45 `abstract`, `public`, `protected`, `internal`, `private`, `virtual`, `override`, or `static`.

46 [Example: The example

```

47     public delegate void StringListEvent(IStringList sender);

```

```

1     public interface IListString
2     {
3         void Add(string s);
4         int Count { get; }
5         event StringListEvent Changed;
6         string this[int index] { get; set; }
7     }

```

8 declares an interface that contains one each of the possible kinds of members: A method, a property, an
9 event, and an indexer. *end example*]

10 An *interface-declaration* creates a new declaration space (§10.3), and the *interface-member-declarations*
11 immediately contained by the *interface-declaration* introduce new members into this declaration space. The
12 following rules apply to *interface-member-declarations*:

- 13 • The name of a method must differ from the names of all properties and events declared in the same
14 interface. In addition, the signature (§10.6) of a method must differ from the signatures of all other
15 methods declared in the same interface.
- 16 • The name of a property or event must differ from the names of all other members declared in the same
17 interface.
- 18 • The signature of an indexer must differ from the signatures of all other indexers declared in the same
19 interface.

20 The inherited members of an interface are specifically not part of the declaration space of the interface.
21 Thus, an interface is allowed to declare a member with the same name or signature as an inherited member.
22 When this occurs, the derived interface member is said to *hide* the base interface member. Hiding an
23 inherited member is not considered an error, but it does cause the compiler to issue a warning. To suppress
24 the warning, the declaration of the derived interface member must include a *new* modifier to indicate that the
25 derived member is intended to hide the base member. This topic is discussed further in §10.7.1.2.

26 If a *new* modifier is included in a declaration that doesn't hide an inherited member, a warning is issued to
27 that effect. This warning is suppressed by removing the *new* modifier.

28 20.2.1 Interface methods

29 Interface methods are declared using *interface-method-declarations*:

```

30     interface-method-declaration:
31     attributesopt newopt return-type identifier ( formal-parameter-listopt ) ;

```

32 The *attributes*, *return-type*, *identifier*, and *formal-parameter-list* of an interface method declaration have the
33 same meaning as those of a method declaration in a class (§17.5). An interface method declaration is not
34 permitted to specify a method body, and the declaration therefore always ends with a semicolon.

35 20.2.2 Interface properties

36 Interface properties are declared using *interface-property-declarations*:

```

37     interface-property-declaration:
38     attributesopt newopt type identifier { interface-accessors }
39
40     interface-accessors:
41     attributesopt get ;
42     attributesopt set ;
43     attributesopt get ; attributesopt set ;
44     attributesopt set ; attributesopt get ;

```

44 The *attributes*, *type*, and *identifier* of an interface property declaration have the same meaning as those of a
45 property declaration in a class (§17.6).

1 The accessors of an interface property declaration correspond to the accessors of a class property declaration
 2 (§17.6.2), except that the accessor body must always be a semicolon. Thus, the accessors simply indicate
 3 whether the property is read-write, read-only, or write-only.

4 **20.2.3 Interface events**

5 Interface events are declared using *interface-event-declarations*:

```
6     interface-event-declaration:
7         attributesopt newopt event type identifier ;
```

8 The *attributes*, *type*, and *identifier* of an interface event declaration have the same meaning as those of an
 9 event declaration in a class (§17.7).

10 **20.2.4 Interface indexers**

11 Interface indexers are declared using *interface-indexer-declarations*:

```
12     interface-indexer-declaration:
13         attributesopt newopt type this [ formal-parameter-list ] { interface-accessors }
```

14 The *attributes*, *type*, and *formal-parameter-list* of an interface indexer declaration have the same meaning as
 15 those of an indexer declaration in a class (§17.8).

16 The accessors of an interface indexer declaration correspond to the accessors of a class indexer declaration
 17 (§17.8), except that the accessor body must always be a semicolon. Thus, the accessors simply indicate
 18 whether the indexer is read-write, read-only, or write-only.

19 **20.2.5 Interface member access**

20 Interface members are accessed through member access (§14.5.4) and indexer access (§14.5.6.2) expressions
 21 of the form *I.M* and *I[A]*, where *I* is an instance of an interface type, *M* is a method, property, or event of
 22 that interface type, and *A* is an indexer argument list.

23 For interfaces that are strictly single-inheritance (each interface in the inheritance chain has exactly zero or
 24 one direct base interface), the effects of the member lookup (§14.3), method invocation (§14.5.5.1), and
 25 indexer access (§14.5.6.2) rules are exactly the same as for classes and structs: More derived members hide
 26 less derived members with the same name or signature. However, for multiple-inheritance interfaces,
 27 ambiguities can occur when two or more unrelated base interfaces declare members with the same name or
 28 signature. This section shows several examples of such situations. In all cases, explicit casts can be used to
 29 resolve the ambiguities.

30 [*Example*: In the example

```
31     interface IList
32     {
33         int Count { get; set; }
34     }
35     interface ICounter
36     {
37         void Count(int i);
38     }
39     interface IListCounter: IList, ICounter {}
40     class C
41     {
42         void Test(IListCounter x) {
43             x.Count(1);           // Error
44             x.Count = 1;         // Error
45             ((IList)x).Count = 1; // Ok, invokes IList.Count.set
46             ((ICounter)x).Count(1); // Ok, invokes ICounter.Count
47         }
48     }
```


1 the first two statements cause compile-time errors because the member lookup (§14.3) of `Count` in
 2 `IListCounter` is ambiguous. As illustrated by the example, the ambiguity is resolved by casting `x` to the
 3 appropriate base interface type. Such casts have no run-time costs—they merely consist of viewing the
 4 instance as a less derived type at compile-time. *end example*]

5 [Example: In the example

```

6     interface IInteger
7     {
8         void Add(int i);
9     }
10    interface IDouble
11    {
12        void Add(double d);
13    }
14    interface INumber: IInteger, IDouble {}
15    class C
16    {
17        void Test(INumber n) {
18            n.Add(1);           // Error, both Add methods are applicable
19            n.Add(1.0);        // Ok, only IDouble.Add is applicable
20            ((IInteger)n).Add(1); // Ok, only IInteger.Add is a candidate
21            ((IDouble)n).Add(1); // Ok, only IDouble.Add is a candidate
22        }
23    }

```

24 the invocation `n.Add(1)` is ambiguous because a method invocation (§14.5.5.1) requires all overloaded
 25 candidate methods to be declared in the same type. However, the invocation `n.Add(1.0)` is permitted
 26 because only `IDouble.Add` is applicable. When explicit casts are inserted, there is only one candidate
 27 method, and thus no ambiguity. *end example*]

28 [Example: In the example

```

29    interface IBase
30    {
31        void F(int i);
32    }
33    interface ILeft: IBase
34    {
35        new void F(int i);
36    }
37    interface IRight: IBase
38    {
39        void G();
40    }
41    interface IDerived: ILeft, IRight {}
42    class A
43    {
44        void Test(IDerived d) {
45            d.F(1);           // Invokes ILeft.F
46            ((IBase)d).F(1); // Invokes IBase.F
47            ((ILeft)d).F(1); // Invokes ILeft.F
48            ((IRight)d).F(1); // Invokes IBase.F
49        }
50    }

```

51 the `IBase.F` member is hidden by the `ILeft.F` member. The invocation `d.F(1)` thus selects `ILeft.F`,
 52 even though `IBase.F` appears to not be hidden in the access path that leads through `IRight`.

53 The intuitive rule for hiding in multiple-inheritance interfaces is simply this: If a member is hidden in any
 54 access path, it is hidden in all access paths. Because the access path from `IDerived` to `ILeft` to `IBase`
 55 hides `IBase.F`, the member is also hidden in the access path from `IDerived` to `IRight` to `IBase`. *end*
 56 *example*]

1 **20.3 Fully qualified interface member names**

2 An interface member is sometimes referred to by its *fully qualified name*. The fully qualified name of an
3 interface member consists of the name of the interface in which the member is declared, followed by a dot,
4 followed by the name of the member. The fully qualified name of a member references the interface in
5 which the member is declared. [*Example:* For example, given the declarations

```
6     interface IControl
7     {
8         void Paint();
9     }
10    interface ITextBox: IControl
11    {
12        void SetText(string text);
13    }
```

14 the fully qualified name of `Paint` is `IControl.Paint` and the fully qualified name of `SetText` is
15 `ITextBox.SetText`. In the example above, it is not possible to refer to `Paint` as `ITextBox.Paint`. *end*
16 *example*]

17 When an interface is part of a namespace, the fully qualified name of an interface member includes the
18 namespace name. [*Example:* For example

```
19    namespace System
20    {
21        public interface ICloneable
22        {
23            object Clone();
24        }
25    }
```

26 Here, the fully qualified name of the `Clone` method is `System.ICloneable.Clone`. *end example*]

27 **20.4 Interface implementations**

28 Interfaces may be implemented by classes and structs. To indicate that a class or struct implements an
29 interface, the interface identifier is included in the base class list of the class or struct. [*Example:* For
30 example:

```
31    interface ICloneable
32    {
33        object Clone();
34    }
35    interface IComparable
36    {
37        int CompareTo(object other);
38    }
39    class ListEntry: ICloneable, IComparable
40    {
41        public object Clone() {...}
42        public int CompareTo(object other) {...}
43    }
```

44 *end example*]

45 A class or struct that implements an interface also implicitly implements all of the interface's base
46 interfaces. This is true even if the class or struct doesn't explicitly list all base interfaces in the base class
47 list. [*Example:* For example:

```
48    interface IControl
49    {
50        void Paint();
51    }
```

```

1     interface ITextBox: IControl
2     {
3         void SetText(string text);
4     }
5     class TextBox: ITextBox
6     {
7         public void Paint() {...}
8         public void SetText(string text) {...}
9     }

```

10 Here, class `TextBox` implements both `IControl` and `ITextBox`. *end example*]

11 20.4.1 Explicit interface member implementations

12 For purposes of implementing interfaces, a class or struct may declare *explicit interface member*
13 *implementations*. An explicit interface member implementation is a method, property, event, or indexer
14 declaration that references a fully qualified interface member name. [*Example*: For example

```

15     interface ICloneable
16     {
17         object Clone();
18     }
19     interface IComparable
20     {
21         int CompareTo(object other);
22     }
23     class ListEntry: ICloneable, IComparable
24     {
25         object ICloneable.Clone() {...}
26         int IComparable.CompareTo(object other) {...}
27     }

```

28 Here, `ICloneable.Clone` and `IComparable.CompareTo` are explicit interface member
29 implementations. *end example*]

30 [*Example*: In some cases, the name of an interface member may not be appropriate for the implementing
31 class, in which case the interface member may be implemented using explicit interface member
32 implementation. A class implementing a file abstraction, for example, would likely implement a `Close`
33 member function that has the effect of releasing the file resource, and implement the `Dispose` method of
34 the `IDisposable` interface using explicit interface member implementation:

```

35     interface IDisposable {
36         void Dispose();
37     }
38     class MyFile: IDisposable {
39         void IDisposable.Dispose() {
40             Close();
41         }
42         public void Close() {
43             // Do what's necessary to close the file
44             System.GC.SuppressFinalize(this);
45         }
46     }

```

47 *end example*]

48 It is not possible to access an explicit interface member implementation through its fully qualified name in a
49 method invocation, property access, or indexer access. An explicit interface member implementation can
50 only be accessed through an interface instance, and is in that case referenced simply by its member name.

51 It is a compile-time error for an explicit interface member implementation to include access modifiers, and it
52 is a compile-time error to include the modifiers `abstract`, `virtual`, `override`, or `static`.

53 Explicit interface member implementations have different accessibility characteristics than other members.
54 Because explicit interface member implementations are never accessible through their fully qualified name

1 in a method invocation or a property access, they are in a sense private. However, since they can be accessed
2 through an interface instance, they are in a sense also public.

3 Explicit interface member implementations serve two primary purposes:

- 4 • Because explicit interface member implementations are not accessible through class or struct instances,
5 they allow interface implementations to be excluded from the public interface of a class or struct. This is
6 particularly useful when a class or struct implements an internal interface that is of no interest to a
7 consumer of that class or struct.
- 8 • Explicit interface member implementations allow disambiguation of interface members with the same
9 signature. Without explicit interface member implementations it would be impossible for a class or
10 struct to have different implementations of interface members with the same signature and return type,
11 as would it be impossible for a class or struct to have any implementation at all of interface members
12 with the same signature but with different return types.

13 For an explicit interface member implementation to be valid, the class or struct must name an interface in its
14 base class list that contains a member whose fully qualified name, type, and parameter types exactly match
15 those of the explicit interface member implementation. [*Example:* Thus, in the following class

```
16     class Shape: ICloneable
17     {
18         object ICloneable.Clone() {...}
19         int IComparable.CompareTo(object other) {...} // invalid
20     }
```

21 the declaration of `IComparable.CompareTo` results in a compile-time error because `IComparable` is not
22 listed in the base class list of `Shape` and is not a base interface of `ICloneable`. Likewise, in the
23 declarations

```
24     class Shape: ICloneable
25     {
26         object ICloneable.Clone() {...}
27     }
28     class Ellipse: Shape
29     {
30         object ICloneable.Clone() {...} // invalid
31     }
```

32 the declaration of `ICloneable.Clone` in `Ellipse` results in a compile-time error because `ICloneable` is
33 not explicitly listed in the base class list of `Ellipse`. *end example*]

34 The fully qualified name of an interface member must reference the interface in which the member was
35 declared. [*Example:* Thus, in the declarations

```
36     interface IControl
37     {
38         void Paint();
39     }
40     interface ITextBox: IControl
41     {
42         void SetText(string text);
43     }
44     class TextBox: ITextBox
45     {
46         void IControl.Paint() {...}
47         void ITextBox.SetText(string text) {...}
48     }
```

49 the explicit interface member implementation of `Paint` must be written as `IControl.Paint`. *end*
50 *example*]

1 20.4.2 Interface mapping

2 A class or struct must provide implementations of all members of the interfaces that are listed in the base
3 class list of the class or struct. The process of locating implementations of interface members in an
4 implementing class or struct is known as *interface mapping*.

5 Interface mapping for a class or struct C locates an implementation for each member of each interface
6 specified in the base class list of C. The implementation of a particular interface member $I.M$, where I is the
7 interface in which the member M is declared, is determined by examining each class or struct S , starting with
8 C and repeating for each successive base class of C, until a match is located:

- 9 • If S contains a declaration of an explicit interface member implementation that matches I and M , then
10 this member is the implementation of $I.M$.
- 11 • Otherwise, if S contains a declaration of a non-static public member that matches M , then this member is
12 the implementation of $I.M$.

13 A compile-time error occurs if implementations cannot be located for all members of all interfaces specified
14 in the base class list of C. Note that the members of an interface include those members that are inherited
15 from base interfaces.

16 For purposes of interface mapping, a class member A matches an interface member B when:

- 17 • A and B are methods, and the name, type, and formal parameter lists of A and B are identical.
- 18 • A and B are properties, the name and type of A and B are identical, and A has the same accessors as B (A
19 is permitted to have additional accessors if it is not an explicit interface member implementation).
- 20 • A and B are events, and the name and type of A and B are identical.
- 21 • A and B are indexers, the type and formal parameter lists of A and B are identical, and A has the same
22 accessors as B (A is permitted to have additional accessors if it is not an explicit interface member
23 implementation).

24 Notable implications of the interface-mapping algorithm are:

- 25 • Explicit interface member implementations take precedence over other members in the same class or
26 struct when determining the class or struct member that implements an interface member.
- 27 • Neither non-public nor static members participate in interface mapping.

28 [*Example:* In the example

```
29     interface ICloneable
30     {
31         object Clone();
32     }
33     class C: ICloneable
34     {
35         object ICloneable.Clone() {...}
36         public object Clone() {...}
37     }
```

38 the `ICloneable.Clone` member of C becomes the implementation of `Clone` in `ICloneable` because
39 explicit interface member implementations take precedence over other members. *end example*]

40 If a class or struct implements two or more interfaces containing a member with the same name, type, and
41 parameter types, it is possible to map each of those interface members onto a single class or struct member.

42 [*Example:* For example

```
43     interface IControl
44     {
45         void Paint();
46     }
```

C# LANGUAGE SPECIFICATION

```
1     interface IForm
2     {
3         void Paint();
4     }
5     class Page: IControl, IForm
6     {
7         public void Paint() {...}
8     }
```

9 Here, the `Paint` methods of both `IControl` and `IForm` are mapped onto the `Paint` method in `Page`. It is
10 of course also possible to have separate explicit interface member implementations for the two methods. *end*
11 *example*]

12 If a class or struct implements an interface that contains hidden members, then some members must
13 necessarily be implemented through explicit interface member implementations. [*Example: For example*

```
14     interface IBase
15     {
16         int P { get; }
17     }
18     interface IDerived: IBase
19     {
20         new int P();
21     }
```

22 An implementation of this interface would require at least one explicit interface member implementation,
23 and would take one of the following forms

```
24     class C: IDerived
25     {
26         int IBase.P { get {...} }
27         int IDerived.P() {...}
28     }
29     class C: IDerived
30     {
31         public int P { get {...} }
32         int IDerived.P() {...}
33     }
34     class C: IDerived
35     {
36         int IBase.P { get {...} }
37         public int P() {...}
38     }
```

39 *end example*]

40 When a class implements multiple interfaces that have the same base interface, there can be only one
41 implementation of the base interface. [*Example: In the example*

```
42     interface IControl
43     {
44         void Paint();
45     }
46     interface ITextBox: IControl
47     {
48         void SetText(string text);
49     }
50     interface IListBox: IControl
51     {
52         void SetItems(string[] items);
53     }
54     class ComboBox: IControl, ITextBox, IListBox
55     {
56         void IControl.Paint() {...}
```

```

1         void ITextBox.SetText(string text) {...}
2         void IListBox.SetItems(string[] items) {...}
3     }

```

4 it is not possible to have separate implementations for the `IControl` named in the base class list, the
5 `IControl` inherited by `ITextBox`, and the `IControl` inherited by `IListBox`. Indeed, there is no notion of
6 a separate identity for these interfaces. Rather, the implementations of `ITextBox` and `IListBox` share the
7 same implementation of `IControl`, and `ComboBox` is simply considered to implement three interfaces,
8 `IControl`, `ITextBox`, and `IListBox`. *end example*]

9 The members of a base class participate in interface mapping. [*Example:* In the example

```

10     interface Interface1
11     {
12         void F();
13     }
14     class Class1
15     {
16         public void F() {}
17         public void G() {}
18     }
19     class Class2: Class1, Interface1
20     {
21         new public void G() {}
22     }

```

23 the method `F` in `Class1` is used in `Class2`'s implementation of `Interface1`. *end example*]

24 20.4.3 Interface implementation inheritance

25 A class inherits all interface implementations provided by its base classes.

26 Without explicitly *re-implementing* an interface, a derived class cannot in any way alter the interface
27 mappings it inherits from its base classes. [*Example:* For example, in the declarations

```

28     interface IControl
29     {
30         void Paint();
31     }
32     class Control: IControl
33     {
34         public void Paint() {...}
35     }
36     class TextBox: Control
37     {
38         new public void Paint() {...}
39     }

```

40 the `Paint` method in `TextBox` hides the `Paint` method in `Control`, but it does not alter the mapping of
41 `Control.Paint` onto `IControl.Paint`, and calls to `Paint` through class instances and interface
42 instances will have the following effects

```

43     Control c = new Control();
44     TextBox t = new TextBox();
45     IControl ic = c;
46     IControl it = t;
47     c.Paint();           // invokes Control.Paint();
48     t.Paint();           // invokes TextBox.Paint();
49     ic.Paint();          // invokes Control.Paint();
50     it.Paint();          // invokes Control.Paint();

```

51 *end example*]

1 However, when an interface method is mapped onto a virtual method in a class, it is possible for derived
 2 classes to override the virtual method and alter the implementation of the interface. [*Example:* For example,
 3 rewriting the declarations above to

```

4     interface IControl
5     {
6         void Paint();
7     }
8
9     class Control: IControl
10    {
11        public virtual void Paint() {...}
12    }
13
14    class TextBox: Control
15    {
16        public override void Paint() {...}
17    }

```

16 the following effects will now be observed

```

17    Control c = new Control();
18    TextBox t = new TextBox();
19    IControl ic = c;
20    IControl it = t;
21    c.Paint();           // invokes Control.Paint();
22    t.Paint();           // invokes TextBox.Paint();
23    ic.Paint();          // invokes Control.Paint();
24    it.Paint();          // invokes TextBox.Paint();

```

25 *end example]*

26 Since explicit interface member implementations cannot be declared virtual, it is not possible to override an
 27 explicit interface member implementation. However, it is perfectly valid for an explicit interface member
 28 implementation to call another method, and that other method can be declared virtual to allow derived
 29 classes to override it. [*Example:* For example

```

30    interface IControl
31    {
32        void Paint();
33    }
34
35    class Control: IControl
36    {
37        void IControl.Paint() { PaintControl(); }
38        protected virtual void PaintControl() {...}
39    }
40
41    class TextBox: Control
42    {
43        protected override void PaintControl() {...}
44    }

```

43 Here, classes derived from `Control` can specialize the implementation of `IControl.Paint` by overriding
 44 the `PaintControl` method. *end example]*

45 20.4.4 Interface re-implementation

46 A class that inherits an interface implementation is permitted to *re-implement* the interface by including it in
 47 the base class list.

48 A re-implementation of an interface follows exactly the same interface mapping rules as an initial
 49 implementation of an interface. Thus, the inherited interface mapping has no effect whatsoever on the
 50 interface mapping established for the re-implementation of the interface. [*Example:* For example, in the
 51 declarations


```

1     interface IControl
2     {
3         void Paint();
4     }
5     class Control: IControl
6     {
7         void IControl.Paint() {...}
8     }
9     class MyControl: Control, IControl
10    {
11        public void Paint() {}
12    }

```

13 the fact that `Control` maps `IControl.Paint` onto `Control.IControl.Paint` doesn't affect the re-
14 implementation in `MyControl`, which maps `IControl.Paint` onto `MyControl.Paint`. *end example*

15 Inherited public member declarations and inherited explicit interface member declarations participate in the
16 interface mapping process for re-implemented interfaces. [*Example: For example*

```

17    interface IMethods
18    {
19        void F();
20        void G();
21        void H();
22        void I();
23    }
24    class Base: IMethods
25    {
26        void IMethods.F() {}
27        void IMethods.G() {}
28        public void H() {}
29        public void I() {}
30    }
31    class Derived: Base, IMethods
32    {
33        public void F() {}
34        void IMethods.H() {}
35    }

```

36 Here, the implementation of `IMethods` in `Derived` maps the interface methods onto `Derived.F`,
37 `Base.IMethods.G`, `Derived.IMethods.H`, and `Base.I`. *end example*

38 When a class implements an interface, it implicitly also implements all of that interface's base interfaces.
39 Likewise, a re-implementation of an interface is also implicitly a re-implementation of all of the interface's
40 base interfaces. [*Example: For example*

```

41    interface IBase
42    {
43        void F();
44    }
45    interface IDerived: IBase
46    {
47        void G();
48    }
49    class C: IDerived
50    {
51        void IBase.F() {...}
52        void IDerived.G() {...}
53    }
54    class D: C, IDerived
55    {
56        public void F() {...}
57        public void G() {...}
58    }

```

1 Here, the re-implementation of `IDerived` also re-implements `IBase.F` onto `D.F`. *end*
 2 *example*]

3 **20.4.5 Abstract classes and interfaces**

4 Like a non-abstract class, an abstract class must provide implementations of all members of the interfaces
 5 that are listed in the base class list of the class. However, an abstract class is permitted to map interface
 6 methods onto abstract methods. [*Example: For example*

```
7     interface IMethods
8     {
9         void F();
10        void G();
11    }
12    abstract class C: IMethods
13    {
14        public abstract void F();
15        public abstract void G();
16    }
```

17 Here, the implementation of `IMethods` maps `F` and `G` onto abstract methods, which must be overridden in
 18 non-abstract classes that derive from `C`. *end example*]

19 Note that explicit interface member implementations cannot be abstract, but explicit interface member
 20 implementations are of course permitted to call abstract methods. [*Example: For example*

```
21     interface IMethods
22     {
23         void F();
24         void G();
25    }
26    abstract class C: IMethods
27    {
28        void IMethods.F() { FF(); }
29        void IMethods.G() { GG(); }
30        protected abstract void FF();
31        protected abstract void GG();
32    }
```

33 Here, non-abstract classes that derive from `C` would be required to override `FF` and `GG`, thus providing the
 34 actual implementation of `IMethods`. *end example*]

21. Enums

1

2 An *enum type* is a distinct type that declares a set of named constants. [Example: The example

```
3     enum Color
4     {
5         Red,
6         Green,
7         Blue
8     }
```

9 declares an enum type named `Color` with members `Red`, `Green`, and `Blue`. *end example*]

10 21.1 Enum declarations

11 An enum declaration declares a new enum type. An enum declaration begins with the keyword `enum`, and
12 defines the name, accessibility, underlying type, and members of the enum.

13 *enum-declaration:*

14 *attributes_{opt} enum-modifiers_{opt} enum identifier enum-base_{opt} enum-body ;_{opt}*

15 *enum-base:*

16 *: integral-type*

17 *enum-body:*

18 *{ enum-member-declarations_{opt} }*

19 *{ enum-member-declarations , }*

20 Each enum type has a corresponding integral type called the *underlying type* of the enum type. This
21 underlying type must be able to represent all the enumerator values defined in the enumeration. An enum
22 declaration may explicitly declare an underlying type of `byte`, `sbyte`, `short`, `ushort`, `int`, `uint`, `long`
23 or `ulong`. [Note: `char` cannot be used as an underlying type. *end note*] An enum declaration that does not
24 explicitly declare an underlying type has an underlying type of `int`.

25 [Example: The example

```
26     enum Color: long
27     {
28         Red,
29         Green,
30         Blue
31     }
```

32 declares an enum with an underlying type of `long`. *end example*] [Note: A developer might choose to use an
33 underlying type of `long`, as in the example, to enable the use of values that are in the range of `long` but not
34 in the range of `int`, or to preserve this option for the future. *end note*]

35 21.2 Enum modifiers

36 An *enum-declaration* may optionally include a sequence of enum modifiers:

37 *enum-modifiers:*

38 *enum-modifier*

39 *enum-modifiers enum-modifier*

```

1      enum-modifier:
2          new
3          public
4          protected
5          internal
6          private

```

7 It is a compile-time error for the same modifier to appear multiple times in an enum declaration.

8 The modifiers of an enum declaration have the same meaning as those of a class declaration (§17.1.1). Note,
9 however, that the `abstract` and `sealed` modifiers are not permitted in an enum declaration. Enums cannot
10 be abstract and do not permit derivation.

11 21.3 Enum members

12 The body of an enum type declaration defines zero or more enum members, which are the named constants
13 of the enum type. No two enum members can have the same name.

```

14      enum-member-declarations:
15          enum-member-declaration
16          enum-member-declarations , enum-member-declaration
17
18      enum-member-declaration:
19          attributesopt identifier
20          attributesopt identifier = constant-expression

```

21 Each enum member has an associated constant value. The type of this value is the underlying type for the
22 containing enum. The constant value for each enum member must be in the range of the underlying type for
the enum. [*Example:* The example

```

23      enum Color: uint
24      {
25          Red = -1,
26          Green = -2,
27          Blue = -3
28      }

```

29 results in a compile-time error because the constant values `-1`, `-2`, and `-3` are not in the range of the
30 underlying integral type `uint`. *end example*]

31 Multiple enum members may share the same associated value. [*Example:* The example

```

32      enum Color
33      {
34          Red,
35          Green,
36          Blue,
37
38          Max = Blue
39      }

```

40 shows an enum that has two enum members—`Blue` and `Max`—that have the same associated value. *end*
41 *example*]

42 The associated value of an enum member is assigned either implicitly or explicitly. If the declaration of the
43 enum member has a *constant-expression* initializer, the value of that constant expression, implicitly
44 converted to the underlying type of the enum, is the associated value of the enum member. If the declaration
45 of the enum member has no initializer, its associated value is set implicitly, as follows:

- 46 • If the enum member is the first enum member declared in the enum type, its associated value is zero.
- 47 • Otherwise, the associated value of the enum member is obtained by increasing the associated value of
48 the textually preceding enum member by one. This increased value must be within the range of values
49 that can be represented by the underlying type.

```

1  [Example: The example
2      using System;
3      enum Color
4      {
5          Red,
6          Green = 10,
7          Blue
8      }
9      class Test
10     {
11         static void Main() {
12             Console.WriteLine(StringFromColor(Color.Red));
13             Console.WriteLine(StringFromColor(Color.Green));
14             Console.WriteLine(StringFromColor(Color.Blue));
15         }
16         static string StringFromColor(Color c) {
17             switch (c) {
18                 case Color.Red:
19                     return String.Format("Red = {0}", (int) c);
20                 case Color.Green:
21                     return String.Format("Green = {0}", (int) c);
22                 case Color.Blue:
23                     return String.Format("Blue = {0}", (int) c);
24                 default:
25                     return "Invalid color";
26             }
27         }
28     }

```

29 prints out the enum member names and their associated values. The output is:

```

30     Red = 0
31     Green = 10
32     Blue = 11

```

33 for the following reasons:

- 34 • the enum member Red is automatically assigned the value zero (since it has no initializer and is the first
- 35 enum member);
- 36 • the enum member Green is explicitly given the value 10;
- 37 • and the enum member Blue is automatically assigned the value one greater than the member that
- 38 textually precedes it.

39 *end example]*

40 The associated value of an enum member may not, directly or indirectly, use the value of its own associated
41 enum member. Other than this circularity restriction, enum member initializers may freely refer to other
42 enum member initializers, regardless of their textual position. Within an enum member initializer, values of
43 other enum members are always treated as having the type of their underlying type, so that casts are not
44 necessary when referring to other enum members.

45 [Example: The example

```

46     enum Circular
47     {
48         A = B,
49         B
50     }

```

51 results in a compile-time error because the declarations of A and B are circular. A depends on B explicitly,
52 and B depends on A implicitly. *end example]*

1 Enum members are named and scoped in a manner exactly analogous to fields within classes. The scope of
2 an enum member is the body of its containing enum type. Within that scope, enum members can be referred
3 to by their simple name. From all other code, the name of an enum member must be qualified with the name
4 of its enum type. Enum members do not have any declared accessibility—an enum member is accessible if
5 its containing enum type is accessible.

6 **21.4 Enum values and operations**

7 Each enum type defines a distinct type; an explicit enumeration conversion (§13.2.2) is required to convert
8 between an enum type and an integral type, or between two enum types. The set of values that an enum type
9 can take on is not limited by its enum members. In particular, any value of the underlying type of an enum
10 can be cast to the enum type, and is a distinct valid value of that enum type.

11 Enum members have the type of their containing enum type (except within other enum member initializers:
12 see §21.3). The value of an enum member declared in enum type `E` with associated value `v` is `(E)v`.

13 The following operators can be used on values of enum types: `==`, `!=`, `<`, `>`, `<=`, `>=` (§14.9.5), `+` (§14.7.4),
14 `-` (§14.7.5), `^`, `&`, `|` (§14.10.2), `~` (§14.6.4), `++`, `--` (§14.5.9 and §14.6.5), and `sizeof` (§25.5.4).

15 Every enum type automatically derives from the class `System.Enum`. Thus, inherited methods and
16 properties of this class can be used on values of an enum type.

22. Delegates

1

2 [Note: Delegates enable scenarios that other languages—such as C++, Pascal, and Modula—have addressed
 3 with function pointers. Unlike C++ function pointers, however, delegates are fully object oriented, and
 4 unlike C++ pointers to member functions, delegates encapsulate both an object instance and a method. *end*
 5 *note*]

6 A delegate declaration defines a class that is derived from the class `System.Delegate`. A delegate instance
 7 encapsulates one or more methods, each of which is referred to as a **callable entity**. For instance methods, a
 8 callable entity consists of an instance and a method on that instance. For static methods, a callable entity
 9 consists of just a method. Given a delegate instance and an appropriate set of arguments, one can invoke all
 10 of that delegate instance’s methods with that set of arguments.

11 An interesting and useful property of a delegate instance is that it does not know or care about the classes of
 12 the methods it encapsulates; all that matters is that those methods be compatible (§22.1) with the delegate’s
 13 type. This makes delegates perfectly suited for “anonymous” invocation.

14 22.1 Delegate declarations

15 A *delegate-declaration* is a *type-declaration* (§16.5) that declares a new delegate type.

16 *delegate-declaration:*
 17 `attributesopt delegate-modifiersopt delegate return-type identifier`
 18 `(formal-parameter-listopt) ;`

19 *delegate-modifiers:*
 20 `delegate-modifier`
 21 `delegate-modifiers delegate-modifier`

22 *delegate-modifier:*
 23 `new`
 24 `public`
 25 `protected`
 26 `internal`
 27 `private`

28 It is a compile-time error for the same modifier to appear multiple times in a delegate declaration.

29 The `new` modifier is only permitted on delegates declared within another type, in which case it specifies that
 30 such a delegate hides an inherited member by the same name, as described in §17.2.2.

31 The `public`, `protected`, `internal`, and `private` modifiers control the accessibility of the delegate type.
 32 Depending on the context in which the delegate declaration occurs, some of these modifiers may not be
 33 permitted (§10.5.1).

34 The delegate’s type name is *identifier*.

35 The optional *formal-parameter-list* specifies the parameters of the delegate, and *return-type* indicates the
 36 return type of the delegate. A method and a delegate type are **compatible** if both of the following are true:

- 37 • They have the same number of parameters, with the same types, in the same order, with the same
 38 parameter modifiers.
- 39 • Their *return-types* are the same.

40 Delegate types in C# are name equivalent, not structurally equivalent. [Note: However, instances of two
 41 distinct but structurally equivalent delegate types may compare as equal (§14.9.8). *end note*] Specifically,

1 two different delegate types that have the same parameter lists and return type are considered **different**
 2 delegate types. [*Example:* For example:

```

3     delegate int D1(int i, double d);
4     class A
5     {
6         public static int M1(int a, double b) {...}
7     }
8     class B
9     {
10        delegate int D2(int c, double d);
11        public static int M1(int f, double g) {...}
12        public static void M2(int k, double l) {...}
13        public static int M3(int g) {...}
14        public static void M4(int g) {...}
15    }

```

16 The delegate types D1 and D2 are both compatible with the methods A.M1 and B.M1, since they have the
 17 same return type and parameter list; however, these delegate types are two different types, so they are not
 18 interchangeable. The delegate types D1 and D2 are incompatible with the methods B.M2, B.M3, and B.M4,
 19 since they have different return types or parameter lists. *end example*]

20 The only way to declare a delegate type is via a *delegate-declaration*. A delegate type is a class type that is
 21 derived from System.Delegate. Delegate types are implicitly sealed, so it is not permissible to derive
 22 any type from a delegate type. It is also not permissible to derive a non-delegate class type from
 23 System.Delegate. Note that System.Delegate is not itself a delegate type; it is a class type from which
 24 all delegate types are derived.

25 C# provides special syntax for delegate instantiation and invocation. Except for instantiation, any operation
 26 that can be applied to a class or class instance can also be applied to a delegate class or instance,
 27 respectively. In particular, it is possible to access members of the System.Delegate type via the usual
 28 member access syntax.

29 The set of methods encapsulated by a delegate instance is called an *invocation list*. When a delegate instance
 30 is created (§22.2) from a single method, it encapsulates that method, and its invocation list contains only one
 31 entry. However, when two non-null delegate instances are combined, their invocation lists are
 32 concatenated—in the order left operand then right operand—to form a new invocation list, which contains
 33 two or more entries.

34 Delegates are combined using the binary + (§14.7.4) and += operators (§14.13.2). A delegate can be
 35 removed from a combination of delegates, using the binary - (§14.7.5) and -= operators (§14.13.2).
 36 Delegates can be compared for equality (§14.9.8).

37 [*Example:* The following example shows the instantiation of a number of delegates, and their corresponding
 38 invocation lists:

```

39     delegate void D(int x);
40     class Test
41     {
42         public static void M1(int i) {...}
43         public static void M2(int i) {...}
44     }
45     class Demo
46     {
47         static void Main() {
48             D cd1 = new D(Test.M1); // M1
49             D cd2 = new D(Test.M2); // M2
50             D cd3 = cd1 + cd2;      // M1 + M2
51             D cd4 = cd3 + cd1;      // M1 + M2 + M1
52             D cd5 = cd4 + cd3;      // M1 + M2 + M1 + M1 + M2
53         }
54     }

```


1 When `cd1` and `cd2` are instantiated, they each encapsulate one method. When `cd3` is instantiated, it has an
 2 invocation list of two methods, `M1` and `M2`, in that order. `cd4`'s invocation list contains `M1`, `M2`, and `M1`, in
 3 that order. Finally, `cd5`'s invocation list contains `M1`, `M2`, `M1`, `M1`, and `M2`, in that order.

4 For more examples of combining (as well as removing) delegates, see §22.3. *end example*]

5 22.2 Delegate instantiation

6 An instance of a delegate is created by a *delegate-creation-expression* (§14.5.10.3). The newly created
 7 delegate instance then refers to either:

- 8 • The static method referenced in the *delegate-creation-expression*, or
- 9 • The target object (which cannot be `null`) and instance method referenced in the *delegate-creation-*
 10 *expression*, or
- 11 • Another delegate

12 [*Example*: For example:

```

13     delegate void D(int x);
14     class Test
15     {
16         public static void M1(int i) {...}
17         public void M2(int i)      {...}
18     }
19     class Demo
20     {
21         static void Main() {
22             D cd1 = new D(Test.M1); // static method
23             Test t = new Test();
24             D cd2 = new D(t.M2);    // instance method
25             D cd3 = new D(cd2);     // another delegate
26         }
27     }
  
```

28 *end example*]

29 Once instantiated, delegate instances always refer to the same target object and method. [*Note*: Remember,
 30 when two delegates are combined, or one is removed from another, a new delegate results with its own
 31 invocation list; the invocation lists of the delegates combined or removed remain unchanged. *end note*]

32 22.3 Delegate invocation

33 C# provides special syntax for invoking a delegate. When a non-`null` delegate instance whose invocation
 34 list contains one entry, is invoked, it invokes the one method with the same arguments it was given, and
 35 returns the same value as the referred to method. (See §14.5.5.2 for detailed information on delegate
 36 invocation.) If an exception occurs during the invocation of such a delegate, and that exception is not caught
 37 within the method that was invoked, the search for an exception catch clause continues in the method that
 38 called the delegate, as if that method had directly called the method to which that delegate referred.

39 Invocation of a delegate instance whose invocation list contains multiple entries, proceeds by invoking each
 40 of the methods in the invocation list, synchronously, in order. Each method so called is passed the same set
 41 of arguments as was given to the delegate instance. If such a delegate invocation includes reference
 42 parameters (§17.5.1.2), each method invocation will occur with a reference to the same variable; changes to
 43 that variable by one method in the invocation list will be visible to methods further down the invocation list.
 44 If the delegate invocation includes output parameters or a return value, their final value will come from the
 45 invocation of the last delegate in the list. If an exception occurs during processing of the invocation of such a
 46 delegate, and that exception is not caught within the method that was invoked, the search for an exception
 47 catch clause continues in the method that called the delegate, and any methods further down the invocation
 48 list are **not** invoked.

C# LANGUAGE SPECIFICATION

1 Attempting to invoke a delegate instance whose value is null results in an exception of type
2 `System.NullReferenceException`.

3 *[Example:* The following example shows how to instantiate, combine, remove, and invoke delegates:

```
4     using System;
5     delegate void D(int x);
6     class Test
7     {
8         public static void M1(int i) {
9             Console.WriteLine("Test.M1: " + i);
10        }
11        public static void M2(int i) {
12            Console.WriteLine("Test.M2: " + i);
13        }
14        public void M3(int i) {
15            Console.WriteLine("Test.M3: " + i);
16        }
17    }
18    class Demo
19    {
20        static void Main() {
21            D cd1 = new D(Test.M1);
22            cd1(-1);    // call M1
23            D cd2 = new D(Test.M2);
24            cd2(-2);    // call M2
25            D cd3 = cd1 + cd2;
26            cd3(10);    // call M1 then M2
27
28            cd3 += cd1;
29            cd3(20);    // call M1, M2, then M1
30            Test t = new Test();
31            D cd4 = new D(t.M3);
32            cd3 += cd4;
33            cd3(30);    // call M1, M2, M1, then M3
34            cd3 -= cd1; // remove last M1
35            cd3(40);    // call M1, M2, then M3
36            cd3 -= cd4;
37            cd3(50);    // call M1 then M2
38            cd3 -= cd2;
39            cd3(60);    // call M1
40            cd3 -= cd2; // impossible removal is benign
41            cd3(60);    // call M1
42            cd3 -= cd1; // invocation list is empty
43            // cd3(70); // System.NullReferenceException thrown
44            cd3 -= cd1; // impossible removal is benign
45        }
46    }
```

47 As shown in the statement `cd3 += cd1;`, a delegate can be present in an invocation list multiple times. In
48 this case, it is simply invoked once per occurrence. In an invocation list such as this, when that delegate is
49 removed, the last occurrence in the invocation list is the one actually removed.

50 Immediately prior to the execution of the final statement, `cd3 -= cd1;`, the delegate `cd3` refers to an
51 empty invocation list. Attempting to remove a delegate from an empty list (or to remove a non-existent
52 delegate from a non-empty list) is not an error.

53 The output produced is:

```
54     Test.M1: -1
55     Test.M2: -2
```

```
1      Test.M1: 10
2      Test.M2: 10
3      Test.M1: 20
4      Test.M2: 20
5      Test.M1: 20
6      Test.M1: 30
7      Test.M2: 30
8      Test.M1: 30
9      Test.M3: 30
10     Test.M1: 40
11     Test.M2: 40
12     Test.M3: 40
13     Test.M1: 50
14     Test.M2: 50
15     Test.M1: 60
16     Test.M1: 60
17     end example]
```


23. Exceptions

1

2 Exceptions in C# provide a structured, uniform, and type-safe way of handling both system level and
 3 application-level error conditions. [Note: The exception mechanism in C# is quite similar to that of C++,
 4 with a few important differences:

- 5 • In C#, all exceptions must be represented by an instance of a class type derived from
 6 `System.Exception`. In C++, any value of any type can be used to represent an exception.
- 7 • In C#, a finally block (§15.10) can be used to write termination code that executes in both normal
 8 execution and exceptional conditions. Such code is difficult to write in C++ without duplicating code.
- 9 • In C#, system-level exceptions such as overflow, divide-by-zero, and null dereferences have well
 10 defined exception classes and are on a par with application-level error conditions.

11 *end note]*

12 23.1 Causes of exceptions

13 Exception can be thrown in two different ways.

- 14 • A `throw` statement (§15.9.5) throws an exception immediately and unconditionally. Control never
 15 reaches the statement immediately following the `throw`.
- 16 • Certain exceptional conditions that arise during the processing of C# statements and expressions cause an
 17 exception in certain circumstances when the operation cannot be completed normally. For example, an
 18 integer division operation (§14.7.2) throws a `System.DivideByZeroException` if the denominator is
 19 zero. See §23.4 for a list of the various exceptions that can occur in this way.

20 23.2 The `System.Exception` class

21 The `System.Exception` class is the base type of all exceptions. This class has a few notable properties
 22 that all exceptions share:

- 23 • `Message` is a read-only property of type `string` that contains a human-readable description of the
 24 reason for the exception.
- 25 • `InnerException` is a read-only property of type `Exception`. If its value is non-`null`, it refers to the
 26 exception that caused the current exception. (That is, the current exception was raised in a `catch` block
 27 handling the type `InnerException`.) Otherwise, its value is `null`, indicating that this exception was not
 28 caused by another exception. (The number of exception objects chained together in this manner can be
 29 arbitrary.)

30 The value of these properties can be specified in calls to the instance constructor for `System.Exception`.

31 23.3 How exceptions are handled

32 Exceptions are handled by a `try` statement (§15.10).

33 When an exception occurs, the system searches for the nearest `catch` clause that can handle the exception,
 34 as determined by the run-time type of the exception. First, the current method is searched for a lexically
 35 enclosing `try` statement, and the associated `catch` clauses of the `try` statement are considered in order. If
 36 that fails, the method that called the current method is searched for a lexically enclosing `try` statement that
 37 encloses the point of the call to the current method. This search continues until a `catch` clause is found that
 38 can handle the current exception, by naming an exception class that is of the same class, or a base class, of

1 the run-time type of the exception being thrown. A `catch` clause that doesn't name an exception class can
 2 handle any exception.

3 Once a matching `catch` clause is found, the system prepares to transfer control to the first statement of the
 4 `catch` clause. Before execution of the `catch` clause begins, the system first executes, in order any
 5 `finally` clauses that were associated with `try` statements more nested than the one that caught the
 6 exception.

7 If no matching `catch` clause is found, one of two things occurs:

- 8 • If the search for a matching `catch` clause reaches a static constructor (§17.11) or static field initializer,
 9 then a `System.TypeInitializationException` is thrown at the point that triggered the invocation
 10 of the static constructor. The inner exception of the `System.TypeInitializationException`
 11 contains the exception that was originally thrown.
- 12 • If the search for matching `catch` clauses reaches the code that initially started the thread, then execution
 13 of the thread is terminated. The impact of such termination is implementation-defined.

14 Exceptions that occur during destructor execution are worth special mention. If an exception occurs during
 15 destructor execution, and that exception is not caught, then the execution of that destructor is terminated and
 16 the destructor of the base class (if any) is called. If there is no base class (as in the case of the `object` type)
 17 or if there is no base class destructor, then the exception is discarded.

18 **23.4 Common Exception Classes**

19 The following exceptions are thrown by certain C# operations.

<code>System.ArithmeticException</code>	A base class for exceptions that occur during arithmetic operations, such as <code>System.DivideByZeroException</code> and <code>System.OverflowException</code> .
<code>System.ArrayTypeMismatchException</code>	Thrown when a store into an array fails because the actual type of the stored element is incompatible with the actual type of the array.
<code>System.DivideByZeroException</code>	Thrown when an attempt to divide an integral value by zero occurs.
<code>System.IndexOutOfRangeException</code>	Thrown when an attempt to index an array via an index that is less than zero or outside the bounds of the array.
<code>System.InvalidCastException</code>	Thrown when an explicit conversion from a base type or interface to a derived type fails at run time.
<code>System.NullReferenceException</code>	Thrown when a <code>null</code> reference is used in a way that causes the referenced object to be required.
<code>System.OutOfMemoryException</code>	Thrown when an attempt to allocate memory (via <code>new</code>) fails.
<code>System.OverflowException</code>	Thrown when an arithmetic operation in a checked context overflows.
<code>System.StackOverflowException</code>	Thrown when the execution stack is exhausted by having too many pending method calls; typically indicative of very deep or unbounded recursion.
<code>System.TypeInitializationException</code>	Thrown when a static constructor throws an exception, and no <code>catch</code> clauses exists to catch it.

24. Attributes

1

2 [Note: Much of the C# language enables the programmer to specify declarative information about the
3 entities defined in the program. For example, the accessibility of a method in a class is specified by
4 decorating it with the *method-modifiers* `public`, `protected`, `internal`, and `private`. *end note*]

5 C# enables programmers to invent new kinds of declarative information, called *attributes*. Programmers can
6 then attach attributes to various program entities, and retrieve attribute information in a run-time
7 environment. [Note: For instance, a framework might define a `HelpAttribute` attribute that can be placed
8 on certain program elements (such as classes and methods) to provide a mapping from those program
9 elements to their documentation. *end note*]

10 Attributes are defined through the declaration of attribute classes (§24.1), which may have positional and
11 named parameters (§24.1.2). Attributes are attached to entities in a C# program using attribute specifications
12 (§24.2), and can be retrieved at run-time as attribute instances (§24.3).

13 24.1 Attribute classes

14 A class that derives from the abstract class `System.Attribute`, whether directly or indirectly, is an
15 *attribute class*. The declaration of an attribute class defines a new kind of attribute that can be placed on a
16 declaration. By convention, attribute classes are named with a suffix of `Attribute`. Uses of an attribute
17 may either include or omit this suffix.

18 24.1.1 Attribute usage

19 The attribute `AttributeUsage` (§24.4.1) is used to describe how an attribute class can be used.

20 `AttributeUsage` has a positional parameter (§24.1.2) that enables an attribute class to specify the kinds of
21 declarations on which it can be used. [Example: The example

```
22     using System;
23     [AttributeUsage(AttributeTargets.Class | AttributeTargets.Interface)]
24     public class SimpleAttribute: Attribute
25     {}
```

26 defines an attribute class named `SimpleAttribute` that can be placed on *class-declarations* and *interface-*
27 *declarations* only. The example

```
28     [Simple] class Class1 {...}
29     [Simple] interface Interface1 {...}
```

30 shows several uses of the `Simple` attribute. Although this attribute is defined with the name
31 `SimpleAttribute`, when this attribute is used, the `Attribute` suffix may be omitted, resulting in the
32 short name `Simple`. Thus, the example above is semantically equivalent to the following

```
33     [SimpleAttribute] class Class1 {...}
34     [SimpleAttribute] interface Interface1 {...}
```

35 *end example*]

36 `AttributeUsage` has a named parameter (§24.1.2), called `AllowMultiple`, which indicates whether the
37 attribute can be specified more than once for a given entity. If `AllowMultiple` for an attribute class is true,
38 then that class is a *multi-use attribute class*, and can be specified more than once on an entity. If
39 `AllowMultiple` for an attribute class is false or it is unspecified, then that class is a *single-use attribute*
40 *class*, and can be specified at most once on an entity.

41 [Example: The example

```

1      using System;
2      [AttributeUsage(AttributeTargets.Class, AllowMultiple = true)]
3      public class AuthorAttribute: Attribute {
4          public AuthorAttribute(string name) {
5              this.name = name;
6          }
7          public string Name { get { return name;} }
8          private string name;
9      }

```

10 defines a multi-use attribute class named `AuthorAttribute`. The example

```

11      [Author("Brian Kernighan"), Author("Dennis Ritchie")]
12      class Class1 {...}

```

13 shows a class declaration with two uses of the `Author` attribute. *end example*

14 `AttributeUsage` has another named parameter (§24.1.2), called `Inherited`, which indicates whether the
15 attribute, when specified on a base class, is also inherited by classes that derive from that base class. If
16 `Inherited` for an attribute class is true, then that attribute is inherited. If `Inherited` for an attribute class
17 is false or it is unspecified, then that attribute is not inherited.

18 An attribute class `X` not having an `AttributeUsage` attribute attached to it, as in

```

19      using System;
20      class X: Attribute { ... }

```

21 is equivalent to the following:

```

22      using System;
23      [AttributeUsage(AttributeTargets.All, AllowMultiple = false, Inherited =
24      true)]
25      class X: Attribute { ... }

```

26 24.1.2 Positional and named parameters

27 Attribute classes can have *positional parameters* and *named parameters*. Each public instance constructor
28 for an attribute class defines a valid sequence of positional parameters for that attribute class. Each non-
29 static public read-write field and property for an attribute class defines a named parameter for the attribute
30 class.

31 [*Example:* The example

```

32      using System;
33      [AttributeUsage(AttributeTargets.Class)]
34      public class HelpAttribute: Attribute
35      {
36          public HelpAttribute(string url) { // url is a positional parameter
37              ...
38          }
39          public string Topic { // Topic is a named parameter
40              get {...}
41              set {...}
42          }
43          public string Url { get {...} }
44      }

```

45 defines an attribute class named `HelpAttribute` that has one positional parameter (`string url`) and one
46 named parameter (`string Topic`). Although it is non-static and public, the property `Url` does not define a
47 named parameter, since it is not read-write.

48 This attribute class might be used as follows:

```

49      [Help("http://www.mycompany.com/.../Class1.htm")]
50      class Class1 {
51      }

```



```

1      [Help("http://www.mycompany.com/.../Misc.htm", Topic ="Class2")]
2      class Class2 {
3      }

```

4 *end example*]

5 24.1.3 Attribute parameter types

6 The types of positional and named parameters for an attribute class are limited to the *attribute parameter*
7 *types*, which are:

- 8 • One of the following types: `bool`, `byte`, `char`, `double`, `float`, `int`, `long`, `short`, `string`.
- 9 • The type `object`.
- 10 • The type `System.Type`.
- 11 • An enum type, provided it has public accessibility and the types in which it is nested (if any) also have
12 public accessibility.
- 13 • Single-dimensional arrays of the above types.

14 24.2 Attribute specification

15 *Attribute specification* is the application of a previously defined attribute to a declaration. An attribute is a
16 piece of additional declarative information that is specified for a declaration. Attributes can be specified at
17 global scope (to specify attributes on the containing assembly) and for *type-declarations* (§16.5), *class-*
18 *member-declarations* (§17.2), *interface-member-declarations* (§20.2), *enum-member-declarations* (§21.1),
19 *accessor-declarations* (§17.6.2), *event-accessor-declarations* (§17.7), and *formal-parameter-lists* (§17.5.1).

20 Attributes are specified in *attribute sections*. An attribute section consists of a pair of square brackets, which
21 surround a comma-separated list of one or more attributes. The order in which attributes are specified in
22 such a list, and the order in which sections attached to the same program entity are arranged, is not
23 significant. For instance, the attribute specifications `[A][B]`, `[B][A]`, `[A, B]`, and `[B, A]` are equivalent.

24 *global-attributes:*

25 *global-attribute-sections*

26 *global-attribute-sections:*

27 *global-attribute-section*

28 *global-attribute-sections global-attribute-section*

29 *global-attribute-section:*

30 [*global-attribute-target-specifier attribute-list*]

31 [*global-attribute-target-specifier attribute-list* ,]

32 *global-attribute-target-specifier:*

33 *global-attribute-target* :

34 *global-attribute-target:*

35 `assembly`

36 *attributes:*

37 *attribute-sections*

38 *attribute-sections:*

39 *attribute-section*

40 *attribute-sections attribute-section*

41 *attribute-section:*

42 [*attribute-target-specifier_{opt} attribute-list*]

43 [*attribute-target-specifier_{opt} attribute-list* ,]

44 *attribute-target-specifier:*

45 *attribute-target* :

```

1      attribute-target:
2          field
3          event
4          method
5          param
6          property
7          return
8          type
9
9      attribute-list:
10         attribute
11         attribute-list , attribute
12
12     attribute:
13         attribute-name attribute-argumentsopt
14
14     attribute-name:
15         type-name
16
16     attribute-arguments:
17         ( positional-argument-listopt )
18         ( positional-argument-list , named-argument-list )
19         ( named-argument-list )
20
20     positional-argument-list:
21         positional-argument
22         positional-argument-list , positional-argument
23
23     positional-argument:
24         attribute-argument-expression
25
25     named-argument-list:
26         named-argument
27         named-argument-list , named-argument
28
28     named-argument:
29         identifier = attribute-argument-expression
30
30     attribute-argument-expression:
31         expression

```

32 An attribute consists of an *attribute-name* and an optional list of positional and named arguments. The
33 positional arguments (if any) precede the named arguments. A positional argument consists of an *attribute-*
34 *argument-expression*; a named argument consists of a name, followed by an equal sign, followed by an
35 *attribute-argument-expression*, which, together, are constrained by the same rules as simple assignment.)
36 The order of named arguments is not significant.

37 The *attribute-name* identifies an attribute class. If the form of *attribute-name* is *type-name* then this name
38 must refer to an attribute class. Otherwise, a compile-time error occurs. [*Example*: The example

```

39     class Class1 {}
40     [Class1] class Class2 {} // Error

```

41 results in a compile-time error because it attempts to use Class1 as an attribute class when Class1 is not
42 an attribute class. *end example*]

43 Certain contexts permit the specification of an attribute on more than one target. A program can explicitly
44 specify the target by including an *attribute-target-specifier*. When an attribute is placed at the global level, a
45 *global-attribute-target-specifier* is required. In all other locations, a reasonable default is applied, but an
46 *attribute-target-specifier* can be used to affirm or override the default in certain ambiguous cases (or to just
47 affirm the default in non-ambiguous cases). Thus, typically, *attribute-target-specifiers* can be omitted
48 except at the global level. The potentially ambiguous contexts are resolved as follows:

- 1 • An attribute specified on a delegate declaration can apply either to the delegate being declared or to its
2 return value. In the absence of an *attribute-target-specifier*, the attribute applies to the delegate. The
3 presence of the `type` *attribute-target-specifier* indicates that the attribute applies to the delegate; the
4 presence of the `return` *attribute-target-specifier* indicates that the attribute applies to the return value.
- 5 • An attribute specified on a method declaration can apply either to the method being declared or to its
6 return value. In the absence of an *attribute-target-specifier*, the attribute applies to the method. The
7 presence of the `method` *attribute-target-specifier* indicates that the attribute applies to the method; the
8 presence of the `return` *attribute-target-specifier* indicates that the attribute applies to the return value.
- 9 • An attribute specified on an operator declaration can apply either to the operator being declared or to its
10 return value of this declaration. In the absence of an *attribute-target-specifier*, the attribute applies to the
11 operator. The presence of the `type` *attribute-target-specifier* indicates that the attribute applies to the
12 operator; the presence of the `return` *attribute-target-specifier* indicates that the attribute applies to the
13 return value.
- 14 • An attribute specified on an event declaration that omits event accessors can apply to the event being
15 declared, to the associated field (if the event is not abstract), or to the associated add and remove
16 methods. In the absence of an *attribute-target-specifier*, the attribute applies to the event declaration.
17 The presence of the `event` *attribute-target-specifier* indicates that the attribute applies to the event; the
18 presence of the `field` *attribute-target-specifier* indicates that the attribute applies to the field; and the
19 presence of the `method` *attribute-target-specifier* indicates that the attribute applies to the methods.
- 20 • An attribute specified on a get accessor declaration for a property or indexer declaration can apply either
21 to the associated method or to its return value. In the absence of an *attribute-target-specifier*, the
22 attribute applies to the method. The presence of the `method` *attribute-target-specifier* indicates that the
23 attribute applies to the method; the presence of the `return` *attribute-target-specifier* indicates that the
24 attribute applies to the return value.
- 25 • An attribute specified on a set accessor for a property or indexer declaration can apply either to the
26 associated method or to its lone implicit parameter. In the absence of an *attribute-target-specifier*, the
27 attribute applies to the method. The presence of the `method` *attribute-target-specifier* indicates that the
28 attribute applies to the method; the presence of the `param` *attribute-target-specifier* indicates that the
29 attribute applies to the parameter.
- 30 • An attribute specified on an add or remove accessor declaration for an event declaration can apply either
31 to the associated method or to its lone parameter. In the absence of an *attribute-target-specifier*, the
32 attribute applies to the method. The presence of the `method` *attribute-target-specifier* indicates that the
33 attribute applies to the method; the presence of the `param` *attribute-target-specifier* indicates that the
34 attribute applies to the parameter.

35 An implementation may accept other attribute target specifiers, the purpose of which is implementation-
36 defined. However, an implementation that does not recognize such a target, shall issue a warning.

37 By convention, attribute classes are named with a suffix of `Attribute`. An *attribute-name* of the form *type-*
38 *name* may either include or omit this suffix. If an attribute class is found both with and without this suffix,
39 an ambiguity is present, and a compile-time error shall be issued. If the *attribute-name* is spelled using a
40 verbatim identifier (§9.4.2), then only an attribute without a suffix is matched, thus enabling such an
41 ambiguity to be resolved. [Example: The example

```
42     using System;
43     [AttributeUsage(AttributeTargets.All)]
44     public class X: Attribute
45     {}
46
47     [AttributeUsage(AttributeTargets.All)]
48     public class XAttribute: Attribute
49     {}
49     [X]
50     class Class1 {} // error: ambiguity
```

C# LANGUAGE SPECIFICATION

```
1      [XAttribute]      // refers to XAttribute
2      class Class2 {}
3      [@X]              // refers to X
4      class Class3 {}
5      [@XAttribute]    // refers to XAttribute
6      class Class4 {}
```

7 shows two attribute classes named `X` and `XAttribute`. The attribute `[X]` is ambiguous, since it could refer
8 to either `X` or `XAttribute`. Using a verbatim identifier allows the exact intent to be specified in such rare
9 cases. The attribute `[XAttribute]` is not ambiguous (although it would be if there was an attribute class
10 named `XAttributeAttribute!`). If the declaration for class `X` is removed, then both attributes refer to the
11 attribute class named `XAttribute`, as follows:

```
12      using System;
13      [AttributeUsage(AttributeTargets.All)]
14      public class XAttribute: Attribute
15      {}
16      [X]              // refers to XAttribute
17      class Class1 {}
18      [XAttribute]    // refers to XAttribute
19      class Class2 {}
20      [@X]            // error: no attribute named "X"
21      class Class3 {}
```

22 *end example]*

23 It is a compile-time error to use a single-use attribute class more than once on the same entity. [*Example:*
24 The example

```
25      using System;
26      [AttributeUsage(AttributeTargets.Class)]
27      public class HelpStringAttribute: Attribute
28      {
29          string value;
30          public HelpStringAttribute(string value) {
31              this.value = value;
32          }
33          public string value { get {...} }
34      }
35      [HelpString("Description of Class1")]
36      [HelpString("Another description of Class1")]
37      public class Class1 {}
```

38 results in a compile-time error because it attempts to use `HelpString`, which is a single-use attribute class,
39 more than once on the declaration of `Class1`. *end example]*

40 An expression `E` is an *attribute-argument-expression* if all of the following statements are true:

- 41 • The type of `E` is an attribute parameter type (§24.1.3).
- 42 • At compile-time, the value of `E` can be resolved to one of the following:
 - 43 ○ A constant value.
 - 44 ○ A `System.Type` object.
 - 45 ○ A one-dimensional array of *attribute-argument-expressions*.

46 [*Example:* For example:

```

1      using System;
2      [AttributeUsage(AttributeTargets.Class)]
3      public class MyAttribute: Attribute
4      {
5          public int P1 {
6              get {...}
7              set {...}
8          }
9
10         public Type P2 {
11             get {...}
12             set {...}
13         }
14
15         public object P3 {
16             get {...}
17             set {...}
18         }
19     }
20
21     [My(P1 = 1234, P3 = new int[]{1, 3, 5}, P2 = typeof(float))]
22     class MyClass {}
23
24 end example]

```

24.3 Attribute instances

An *attribute instance* is an instance that represents an attribute at run-time. An attribute is defined with an attribute class, positional arguments, and named arguments. An attribute instance is an instance of the attribute class that is initialized with the positional and named arguments.

Retrieval of an attribute instance involves both compile-time and run-time processing, as described in the following sections.

24.3.1 Compilation of an attribute

The compilation of an *attribute* with attribute class *T*, *positional-argument-list P* and *named-argument-list N*, consists of the following steps:

- 30 • Follow the compile-time processing steps for compiling an *object-creation-expression* of the form
31 `new T(P)`. These steps either result in a compile-time error, or determine an instance constructor on *T*
32 that can be invoked at run-time. Call this instance constructor *C*.
- 33 • If *C* does not have public accessibility, then a compile-time error occurs.
- 34 • For each *named-argument Arg* in *N*:
 - 35 ○ Let *Name* be the *identifier* of the *named-argument Arg*.
 - 36 ○ *Name* must identify a non-static read-write public field or property on *T*. If *T* has no such field or
37 property, then a compile-time error occurs.
- 38 • Keep the following information for run-time instantiation of the attribute: the attribute class *T*, the
39 instance constructor *C* on *T*, the *positional-argument-list P* and the *named-argument-list N*.

24.3.2 Run-time retrieval of an attribute instance

Compilation of an *attribute* yields an attribute class *T*, an instance constructor *C* on *T*, a *positional-argument-list P*, and a *named-argument-list N*. Given this information, an attribute instance can be retrieved at run-time using the following steps:

- 44 • Follow the run-time processing steps for executing an *object-creation-expression* of the form
45 `new T(P)`, using the instance constructor *C* as determined at compile-time. These steps either result in
46 an exception, or produce an instance of *T*. Call this instance *O*.
- 47 • For each *named-argument Arg* in *N*, in order:

- 1 ○ Let *Name* be the *identifier* of the *named-argument* *Arg*. If *Name* does not identify a non-static public
2 read-write field or property on *O*, then an exception is thrown.
- 3 ○ Let *Value* be the result of evaluating the *attribute-argument-expression* of *Arg*.
- 4 ○ If *Name* identifies a field on *O*, then set this field to the value *Value*.
- 5 ○ Otherwise, *Name* identifies a property on *O*. Set this property to the value *Value*.
- 6 ○ The result is *O*, an instance of the attribute class *T* that has been initialized with the *positional-*
7 *argument-list* *P* and the *named-argument-list* *N*.

8 **24.4 Reserved attributes**

9 A small number of attributes affect the language in some way. These attributes include:

- 10 • `System.AttributeUsageAttribute` (§24.4.1), which is used to describe the ways in which an
11 attribute class can be used.
- 12 • `System.ConditionalAttribute` (§24.4.2), which is used to define conditional methods.
- 13 • `System.ObsoleteAttribute` (§24.4.3), which is used to mark a member as obsolete.

14 **24.4.1 The `AttributeUsage` attribute**

15 The attribute `AttributeUsage` is used to describe the manner in which the attribute class can be used.

16 A class that is decorated with the `AttributeUsage` attribute must derive from `System.Attribute`, either
17 directly or indirectly. Otherwise, a compile-time error occurs.

18 [*Note:* For an example of using this attribute, see §24.1.1. *end note*]

19 **24.4.2 The `Conditional` attribute**

20 The attribute `Conditional` enables the definition of *conditional methods*. The `Conditional` attribute
21 indicates a condition by testing a conditional compilation symbol. Calls to a conditional method are either
22 included or omitted depending on whether this symbol is defined at the point of the call. If the symbol is
23 defined, the call is included; otherwise, the call is omitted.

24 A conditional method is subject to the following restrictions:

- 25 • The conditional method must be a method in a *class-declaration*. A compile-time error occurs if the
26 `Conditional` attribute is specified on an interface method.
- 27 • The conditional method must have a return type of `void`.
- 28 • The conditional method must not be marked with the `override` modifier. A conditional method may be
29 marked with the `virtual` modifier, however. Overrides of such a method are implicitly conditional,
30 and must not be explicitly marked with a `Conditional` attribute.
- 31 • The conditional method must not be an implementation of an interface method. Otherwise, a compile-
32 time error occurs.

33 In addition, a compile-time error occurs if a conditional method is used in a *delegate-creation-expression*.

34 [*Example:* The example

35 `#define DEBUG`

```

1      using System;
2      using System.Diagnostics;
3      class Class1
4      {
5          [Conditional("DEBUG")]
6          public static void M() {
7              Console.WriteLine("Executed Class1.M");
8          }
9      }
10     class Class2
11     {
12         public static void Test() {
13             Class1.M();
14         }
15     }

```

16 declares `Class1.M` as a conditional method. `Class2`'s `Test` method calls this method. Since the conditional compilation symbol `DEBUG` is defined, if `Class2.Test` is called, it will call `M`. If the symbol `DEBUG` had not been defined, then `Class2.Test` would not call `Class1.M`. *end example*

19 It is important to note that the inclusion or exclusion of a call to a conditional method is controlled by the conditional compilation symbols at the point of the call. [*Example:* In the example

```

21     // Begin class1.cs
22     using System;
23     using System.Diagnostics;
24     class Class1
25     {
26         [Conditional("DEBUG")]
27         public static void F() {
28             Console.WriteLine("Executed Class1.F");
29         }
30     }
31     // End class1.cs
32
33     // Begin class2.cs
34     #define DEBUG
35     class Class2
36     {
37         public static void G() {
38             Class1.F();           // F is called
39         }
40     }
41     // End class2.cs
42
43     // Begin class3.cs
44     #undef DEBUG
45     class Class3
46     {
47         public static void H() {
48             Class1.F();           // F is not called
49         }
50     }
51     // End class3.cs

```

52 the classes `Class2` and `Class3` each contain calls to the conditional method `Class1.F`, which is conditional based on whether or not `DEBUG` is defined. Since this symbol is defined in the context of `Class2` but not `Class3`, the call to `F` in `Class2` is included, while the call to `F` in `Class3` is omitted. *end example*

55 The use of conditional methods in an inheritance chain can be confusing. Calls made to a conditional method through base, of the form `base.M`, are subject to the normal conditional method call rules. [*Example:* In the example

C# LANGUAGE SPECIFICATION

```
1 // Begin class1.cs
2 using System;
3 using System.Diagnostics;
4 class Class1
5 {
6     [Conditional("DEBUG")]
7     public virtual void M() {
8         Console.WriteLine("Class1.M executed");
9     }
10 }
11 // End class1.cs
12
13 // Begin class2.cs
14 using System;
15 class Class2: Class1
16 {
17     public override void M() {
18         Console.WriteLine("Class2.M executed");
19         base.M(); // base.M is not called!
20     }
21 }
22 // End class2.cs
23
24 // Begin class3.cs
25 #define DEBUG
26 using System;
27 class Class3
28 {
29     public static void Test() {
30         Class2 c = new Class2();
31         c.M(); // M is called
32     }
33 }
34 // End class3.cs
```

35 Class2 includes a call to the M defined in its base class. This call is omitted because the base method is
36 conditional based on the presence of the symbol DEBUG, which is undefined. Thus, the method writes to the
37 console "Class2.M executed" only. Judicious use of *pp-declarations* can eliminate such problems. *end*
38 *example*]

39 24.4.3 The Obsolete attribute

40 The attribute Obsolete is used to mark types and members of types that should no longer be used.

```
41 using System;
42 [AttributeUsage(AttributeTargets.Class | AttributeTargets.Struct |
43 AttributeTargets.Enum | AttributeTargets.Interface |
44 AttributeTargets.Delegate | AttributeTargets.Method |
45 AttributeTargets.Constructor | AttributeTargets.Property |
46 AttributeTargets.Field | AttributeTargets.Event)]
47 public class ObsoleteAttribute: Attribute
48 {
49     public ObsoleteAttribute() {...}
50     public ObsoleteAttribute(string message) {...}
51     public ObsoleteAttribute(string message, bool error) {...}
52     public string Message { get {...} }
53     public bool IsError { get {...} }
54 }
```

55 If a program uses a type or member that is decorated with the Obsolete attribute, then the compiler shall
56 issue a warning or error in order to alert the developer, so the offending code can be fixed. Specifically, the
57 compiler shall issue a warning if no error parameter is provided, or if the error parameter is provided and has
58 the value false. The compiler shall issue a compile-time error if the error parameter is specified and has the
59 value true.


```
1  [Example: In the example
2      [Obsolete("This class is obsolete; use class B instead")]
3      class A
4      {
5          public void F() {}
6      }
7      class B
8      {
9          public void F() {}
10     }
11     class Test
12     {
13         static void Main() {
14             A a = new A(); // warning
15             a.F();
16         }
17     }
```

18 the class A is decorated with the `Obsolete` attribute. Each use of A in `Main` results in a warning that
19 includes the specified message, "This class is obsolete; use class B instead."

20 *end example*]

25. Unsafe code

1

2 An implementation that does not support unsafe code is required to diagnose any usage of the keyword
3 `unsafe`.

4 **This remainder of this clause is conditionally normative.**

5 [Note: The core C# language, as defined in the preceding chapters, differs notably from C and C++ in its
6 omission of pointers as a data type. Instead, C# provides references and the ability to create objects that are
7 managed by a garbage collector. This design, coupled with other features, makes C# a much safer language
8 than C or C++. In the core C# language it is simply not possible to have an uninitialized variable, a
9 “dangling” pointer, or an expression that indexes an array beyond its bounds. Whole categories of bugs that
10 routinely plague C and C++ programs are thus eliminated.

11 While practically every pointer type construct in C or C++ has a reference type counterpart in C#,
12 nonetheless, there are situations where access to pointer types becomes a necessity. For example, interfacing
13 with the underlying operating system, accessing a memory-mapped device, or implementing a time-critical
14 algorithm may not be possible or practical without access to pointers. To address this need, C# provides the
15 ability to write *unsafe code*.

16 In unsafe code it is possible to declare and operate on pointers, to perform conversions between pointers and
17 integral types, to take the address of variables, and so forth. In a sense, writing unsafe code is much like
18 writing C code within a C# program.

19 Unsafe code is in fact a “safe” feature from the perspective of both developers and users. Unsafe code must
20 be clearly marked with the modifier `unsafe`, so developers can’t possibly use unsafe features accidentally,
21 and the execution engine works to ensure that unsafe code cannot be executed in an untrusted environment.
22 *end note*]

23 25.1 Unsafe contexts

24 The unsafe features of C# are available only in *unsafe contexts*. An unsafe context is introduced by including
25 an `unsafe` modifier in the declaration of a type or member, or by employing an *unsafe-statement*:

- 26 • A declaration of a class, struct, interface, or delegate may include an `unsafe` modifier, in which case
27 the entire textual extent of that type declaration (including the body of the class, struct, or interface) is
28 considered an unsafe context.
- 29 • A declaration of a field, method, property, event, indexer, operator, instance constructor, destructor, or
30 static constructor may include an `unsafe` modifier, in which case, the entire textual extent of that
31 member declaration is considered an unsafe context.
- 32 • An *unsafe-statement* enables the use of an unsafe context within a *block*. The entire textual extent of the
33 associated *block* is considered an unsafe context.

34 The associated grammar extensions are shown below. For brevity, ellipses (...) are used to represent
35 productions that appear in preceding chapters.

36 *class-modifier*:

37 ...

38 `unsafe`

39 *struct-modifier*:

40 ...

41 `unsafe`

1 *interface-modifier:*
2 ...
3 **unsafe**

4 *delegate-modifier:*
5 ...
6 **unsafe**

7 *field-modifier:*
8 ...
9 **unsafe**

10 *method-modifier:*
11 ...
12 **unsafe**

13 *property-modifier:*
14 ...
15 **unsafe**

16 *event-modifier:*
17 ...
18 **unsafe**

19 *indexer-modifier:*
20 ...
21 **unsafe**

22 *operator-modifier:*
23 ...
24 **unsafe**

25 *constructor-modifier:*
26 ...
27 **unsafe**

28 *destructor-declaration:*
29 *attributes*_{opt} **extern**_{opt} **unsafe**_{opt} ~ *identifier* () *destructor-body*
30 *attributes*_{opt} **unsafe**_{opt} **extern**_{opt} ~ *identifier* () *destructor-body*

31 *static-constructor-declaration:*
32 *attributes*_{opt} **extern**_{opt} **unsafe**_{opt} **static** *identifier* () *static-constructor-body*
33 *attributes*_{opt} **unsafe**_{opt} **extern**_{opt} **static** *identifier* () *static-constructor-body*

34 *embedded-statement:*
35 ...
36 *unsafe-statement*

37 *unsafe-statement:*
38 **unsafe** *block*

39 [*Example:* In the example

```
40       public unsafe struct Node
41       {
42           public int value;
43           public Node* Left;
44           public Node* Right;
45       }
```

46 the **unsafe** modifier specified in the struct declaration causes the entire textual extent of the struct
47 declaration to become an unsafe context. Thus, it is possible to declare the **Left** and **Right** fields to be of a
48 pointer type. The example above could also be written

```

1     public struct Node
2     {
3         public int Value;
4         public unsafe Node* Left;
5         public unsafe Node* Right;
6     }

```

7 Here, the `unsafe` modifiers in the field declarations cause those declarations to be considered unsafe contexts. *end example*

9 Other than establishing an unsafe context, thus permitting the use of pointer types, the `unsafe` modifier has no effect on a type or a member. [*Example:* In the example

```

11    public class A
12    {
13        public unsafe virtual void F() {
14            char* p;
15            ...
16        }
17    }
18    public class B: A
19    {
20        public override void F() {
21            base.F();
22            ...
23        }
24    }

```

25 the `unsafe` modifier on the `F` method in `A` simply causes the textual extent of `F` to become an unsafe context in which the unsafe features of the language can be used. In the override of `F` in `B`, there is no need to re-specify the `unsafe` modifier—unless, of course, the `F` method in `B` itself needs access to unsafe features.

29 The situation is slightly different when a pointer type is part of the method's signature

```

30    public unsafe class A
31    {
32        public virtual void F(char* p) {...}
33    }
34    public class B: A
35    {
36        public unsafe override void F(char* p) {...}
37    }

```

38 Here, because `F`'s signature includes a pointer type, it can only be written in an unsafe context. However, the unsafe context can be introduced by either making the entire class unsafe, as is the case in `A`, or by including an `unsafe` modifier in the method declaration, as is the case in `B`. *end example*

41 25.2 Pointer types

42 In an unsafe context, a *type* (§11) may be a *pointer-type* as well as a *value-type* or a *reference-type*.

```

43     type:
44         value-type
45         reference-type
46         pointer-type

```

47 A *pointer-type* is written as an *unmanaged-type* or the keyword `void`, followed by a `*` token:

```

48     pointer-type:
49         unmanaged-type *
50         void *
51     unmanaged-type:
52         type

```

C# LANGUAGE SPECIFICATION

1 The type specified before the `*` in a pointer type is called the *referent type* of the pointer type. It represents
2 the type of the variable to which a value of the pointer type points.

3 Unlike references (values of reference types), pointers are not tracked by the garbage collector—the garbage
4 collector has no knowledge of pointers and the data to which they point. For this reason a pointer is not
5 permitted to point to a reference or to a struct that contains references, and the referent type of a pointer must
6 be an *unmanaged-type*.

7 An *unmanaged-type* is any type that isn't a *reference-type* and doesn't contain *reference-type* fields at any
8 level of nesting. In other words, an *unmanaged-type* is one of the following:

- 9 • `sbyte`, `byte`, `short`, `ushort`, `int`, `uint`, `long`, `ulong`, `char`, `float`, `double`, `decimal`, or `bool`.
- 10 • Any *enum-type*.
- 11 • Any *pointer-type*.
- 12 • Any user-defined *struct-type* that contains fields of *unmanaged-types* only.

13 The intuitive rule for mixing of pointers and references is that referents of references (objects) are permitted
14 to contain pointers, but referents of pointers are not permitted to contain references.

15 [Example: Some examples of pointer types are given in the table below:
16

Example	Description
<code>byte*</code>	Pointer to <code>byte</code>
<code>char*</code>	Pointer to <code>char</code>
<code>int**</code>	Pointer to pointer to <code>int</code>
<code>int* []</code>	Single-dimensional array of pointers to <code>int</code>
<code>void*</code>	Pointer to unknown type

17
18 *end example]*

19 For a given implementation, all pointer types must have the same size and representation.

20 [Note: Unlike C and C++, when multiple pointers are declared in the same declaration, in C# the `*` is written
21 along with the underlying type only, not as a prefix punctuator on each pointer name. For example:

22 `int* pi, pj; // NOT as int *pi, *pj;`

23 *end note]*

24 The value of a pointer having type `T*` represents the *address* of a variable of type `T`. The pointer indirection
25 operator `*` (§25.5.1) may be used to access this variable. For example, given a variable `P` of type `int*`, the
26 expression `*P` denotes the `int` variable found at the address contained in `P`.

27 Like an object reference, a pointer may be `null`. Applying the indirection operator to a `null` pointer results
28 in implementation-defined behavior. A pointer with value `null` is represented by all-bits-zero.

29 The `void*` type represents a pointer to an unknown type. Because the referent type is unknown, the
30 indirection operator cannot be applied to a pointer of type `void*`, nor can any arithmetic be performed on
31 such a pointer. However, a pointer of type `void*` can be cast to any other pointer type (and vice versa).

32 Pointer types are a separate category of types. Unlike reference types and value types, pointer types do not
33 inherit from `object` and no conversions exist between pointer types and `object`. In particular, boxing and
34 unboxing (§11.3) are not supported for pointers. However, conversions are permitted between different
35 pointer types and between pointer types and the integral types. This is described in §25.4.

36 A *pointer-type* may be used as the type of a volatile field (§17.4.3).

1 [Note: Although pointers can be passed as `ref` or `out` parameters, doing so can cause undefined behavior,
 2 since the pointer may well be set to point to a local variable which no longer exists when the called method
 3 returns, or the fixed object to which it used to point, is no longer fixed. For example:

```

 4     using System;
 5     class Test
 6     {
 7         static int value = 20;
 8
 9         unsafe static void F(out int* pi1, ref int* pi2) {
10             int i = 10;
11             pi1 = &i;
12
13             fixed (int* pj = &value) {
14                 // ...
15                 pi2 = pj;
16             }
17         }
18
19         static void Main() {
20             int i = 10;
21             unsafe {
22                 int* px1;
23                 int* px2 = &i;
24
25                 F(out px1, ref px2);
26                 Console.WriteLine("*px1 = {0}, *px2 = {1}",
27                     *px1, *px2); // undefined behavior
28             }
29         }
30     }
  
```

31 *end note*

32 A method can return a value of some type, and that type can be a pointer. [Example: For example, when
 33 given a pointer to a contiguous sequence of `ints`, that sequence's element count, and some other `int` value,
 34 the following method returns the address of that value in that sequence, if a match occurs; otherwise it
 35 returns `null`:

```

36     unsafe static int* Find(int* pi, int size, int value) {
37         for (int i = 0; i < size; ++i) {
38             if (*pi == value) {
39                 return pi;
40             }
41             ++pi;
42         }
43         return null;
44     }
  
```

45 *end example*

46 In an unsafe context, several constructs are available for operating on pointers:

- 47 • The `*` operator may be used to perform pointer indirection (§25.5.1).
- 48 • The `->` operator may be used to access a member of a struct through a pointer (§25.5.2).
- 49 • The `[]` operator may be used to index a pointer (§25.5.3).
- 50 • The `&` operator may be used to obtain the address of a variable (§25.5.4).
- 51 • The `++` and `--` operators may be used to increment and decrement pointers (§25.5.5).
- 52 • The `+` and `-` operators may be used to perform pointer arithmetic (§25.5.6).
- 53 • The `==`, `!=`, `<`, `>`, `<=`, and `=>` operators may be used to compare pointers (§25.5.7).
- 54 • The `stackalloc` operator may be used to allocate memory from the call stack (§25.7).

- 1 • The `fixed` statement may be used to temporarily fix a variable so its address can be obtained (§25.6).

2 25.3 Fixed and moveable variables

3 The address-of operator (§25.5.4) and the `fixed` statement (§25.6) divide variables into two categories:
4 *Fixed variables* and *moveable variables*.

5 Fixed variables reside in storage locations that are unaffected by operation of the garbage collector.
6 (Examples of fixed variables include local variables, value parameters, and variables created by
7 dereferencing pointers.) On the other hand, moveable variables reside in storage locations that are subject to
8 relocation or disposal by the garbage collector. (Examples of moveable variables include fields in objects
9 and elements of arrays.)

10 The `&` operator (§25.5.4) permits the address of a fixed variable to be obtained without restrictions.
11 However, because a moveable variable is subject to relocation or disposal by the garbage collector, the
12 address of a moveable variable can only be obtained using a `fixed` statement (§25.6), and that address
13 remains valid only for the duration of that `fixed` statement.

14 In precise terms, a fixed variable is one of the following:

- 15 • A variable resulting from a *simple-name* (§14.5.2) that refers to a local variable or a value parameter.
16 • A variable resulting from a *member-access* (§14.5.4) of the form `V.I`, where `V` is a fixed variable of a
17 *struct-type*.
18 • A variable resulting from a *pointer-indirection-expression* (§25.5.1) of the form `*P`, a *pointer-member-*
19 *access* (§25.5.2) of the form `P->I`, or a *pointer-element-access* (§25.5.3) of the form `P[E]`.

20 All other variables are classified as moveable variables.

21 Note that a static field is classified as a moveable variable. Also note that a `ref` or `out` parameter is
22 classified as a moveable variable, even if the argument given for the parameter is a fixed variable. Finally,
23 note that a variable produced by dereferencing a pointer is always classified as a fixed variable.

24 25.4 Pointer conversions

25 In an unsafe context, the set of available implicit conversions (§13.1) is extended to include the following
26 implicit pointer conversions:

- 27 • From any *pointer-type* to the type `void*`.
28 • From the null type to any *pointer-type*.

29 Additionally, in an unsafe context, the set of available explicit conversions (§13.2) is extended to include the
30 following explicit pointer conversions:

- 31 • From any *pointer-type* to any other *pointer-type*.
32 • From `sbyte`, `byte`, `short`, `ushort`, `int`, `uint`, `long`, or `ulong` to any *pointer-type*.
33 • From any *pointer-type* to `sbyte`, `byte`, `short`, `ushort`, `int`, `uint`, `long`, or `ulong`.

34 Finally, in an unsafe context, the set of standard implicit conversions (§13.3.1) includes the following
35 pointer conversion:

- 36 • From any *pointer-type* to the type `void*`.

37 Conversions between two pointer types never change the actual pointer value. In other words, a conversion
38 from one pointer type to another has no effect on the underlying address given by the pointer.

39 When one pointer type is converted to another, if the resulting pointer is not correctly aligned for the
40 pointed-to type, the behavior is undefined if the result is dereferenced. In general, the concept “correctly
41 aligned” is transitive: if a pointer to type A is correctly aligned for a pointer to type B, which, in turn, is
42 correctly aligned for a pointer to type C, then a pointer to type A is correctly aligned for a pointer to type C.

1 [Example: Consider the following case in which a variable having one type is accessed via a pointer to a
2 different type:

```
3     char c = 'A';
4     char* pc = &c;
5     void* pv = pc;
6     int* pi = (int*)pv;
7     int i = *pi;      // undefined
8     *pi = 123456;    // undefined
```

9 *end example]*

10 When a pointer type is converted to a pointer to byte, the result points to the lowest addressed byte of the
11 variable. Successive increments of the result, up to the size of the variable, yield pointers to the remaining
12 bytes of that variable. [Example: For example, the following method displays each of the eight bytes in a
13 double as a hexadecimal value:

```
14     using System;
15     class Test
16     {
17         static void Main() {
18             double d = 123.456e23;
19             unsafe {
20                 byte* pb = (byte*)&d;
21
22                 for (int i = 0; i < sizeof(double); ++i)
23                     Console.Write(" {0,2:x}", (uint)(*pb++));
24                 Console.WriteLine();
25             }
26         }
27     }
```

28 Of course, the output produced depends on endianness. *end example]*

29 Mappings between pointers and integers are implementation-defined. [Note: However, on 32- and 64-bit
30 CPU architectures with a linear address space, conversions of pointers to or from integral types typically
31 behave exactly like conversions of `uint` or `ulong` values, respectively, to or from those integral types. *end*
32 *note]*

33 25.5 Pointers in expressions

34 In an unsafe context, an expression may yield a result of a pointer type, but outside an unsafe context it is a
35 compile-time error for an expression to be of a pointer type. In precise terms, outside an unsafe context a
36 compile-time error occurs if any *simple-name* (§14.5.2), *member-access* (§14.5.4), *invocation-expression*
37 (§14.5.5), or *element-access* (§14.5.6) is of a pointer type.

38 In an unsafe context, the *primary-no-array-creation-expression* (§14.5) and *unary-expression* (§14.6)
39 productions permit the following additional constructs:

40 *primary-no-array-creation-expression:*

```
41     ...
42     pointer-member-access
43     pointer-element-access
44     sizeof-expression
```

45

46 *unary-expression:*

```
47     ...
48     pointer-indirection-expression
49     addressof-expression
```

50 These constructs are described in the following sections.

51 [Note: The precedence and associativity of the unsafe operators is implied by the grammar. *end note]*

1 25.5.1 Pointer indirection

2 A *pointer-indirection-expression* consists of an asterisk (*) followed by a *unary-expression*.

3 *pointer-indirection-expression*:
4 * *unary-expression*

5 The unary * operator denotes *pointer indirection* and is used to obtain the variable to which a pointer points.
6 The result of evaluating *P, where P is an expression of a pointer type T*, is a variable of type T. It is a
7 compile-time error to apply the unary * operator to an expression of type void* or to an expression that
8 isn't of a pointer type.

9 The effect of applying the unary * operator to a null pointer is implementation-defined. In particular, there
10 is no guarantee that this operation throws a System.NullReferenceException.

11 If an invalid value has been assigned to the pointer, the behavior of the unary * operator is undefined. [Note:
12 Among the invalid values for dereferencing a pointer by the unary * operator are an address inappropriately
13 aligned for the type pointed to (see example in §25.4), and the address of a variable after the end of its
14 lifetime. end note]

15 For purposes of definite assignment analysis, a variable produced by evaluating an expression of the form
16 *P is considered initially assigned (§12.3.1).

17 25.5.2 Pointer member access

18 A *pointer-member-access* consists of a *primary-expression*, followed by a “->” token, followed by an
19 *identifier*.

20 *pointer-member-access*:
21 *primary-expression* -> *identifier*

22 In a pointer member access of the form P->I, P must be an expression of a pointer type other than void*,
23 and I must denote an accessible member of the type to which P points.

24 A pointer member access of the form P->I is evaluated exactly as (*P).I. For a description of the pointer
25 indirection operator (*), see §25.5.1. For a description of the member access operator (.), see §14.5.4.

26 [Example: In the example

```
27     struct Point
28     {
29         public int x;
30         public int y;
31
32         public override string ToString() {
33             return "(" + x + ", " + y + ")";
34         }
35     }
36     using System;
37     class Test
38     {
39         static void Main() {
40             Point point;
41             unsafe {
42                 Point* p = &point;
43                 p->x = 10;
44                 p->y = 20;
45                 Console.WriteLine(p->ToString());
46             }
47         }
48     }
```

48 the -> operator is used to access fields and invoke a method of a struct through a pointer. Because the
49 operation P->I is precisely equivalent to (*P).I, the Main method could equally well have been written:

```

1      using System;
2      class Test
3      {
4          static void Main() {
5              Point point;
6              unsafe {
7                  Point* p = &point;
8                  (*p).x = 10;
9                  (*p).y = 20;
10                 Console.WriteLine((*p).ToString());
11             }
12         }
13     }

```

14 *end example]*

15 25.5.3 Pointer element access

16 A *pointer-element-access* consists of a *primary-no-array-creation-expression* followed by an expression enclosed in “[” and “]”.

18 *pointer-element-access:*

19 *primary-no-array-creation-expression* [*expression*]

20 In a pointer element access of the form P[E], P must be an expression of a pointer type other than void*, and E must be an expression of a type that can be implicitly converted to int, uint, long, or ulong.

22 A pointer element access of the form P[E] is evaluated exactly as *(P + E). For a description of the pointer indirection operator (*), see §25.5.1. For a description of the pointer addition operator (+), see §25.5.6.

24 [*Example:* In the example

```

25     class Test
26     {
27         static void Main() {
28             unsafe {
29                 char* p = stackalloc char[256];
30                 for (int i = 0; i < 256; i++) p[i] = (char)i;
31             }
32         }
33     }

```

34 a pointer element access is used to initialize the character buffer in a for loop. Because the operation P[E] is precisely equivalent to *(P + E), the example could equally well have been written:

```

36     class Test
37     {
38         static void Main() {
39             unsafe {
40                 char* p = stackalloc char[256];
41                 for (int i = 0; i < 256; i++) *(p + i) = (char)i;
42             }
43         }
44     }

```

45 *end example]*

46 The pointer element access operator does not check for out-of-bounds errors and the behavior when accessing an out-of-bounds element is undefined. [*Note:* This is the same as C and C++. *end note]*

48 25.5.4 The address-of operator

49 An *addressof-expression* consists of an ampersand (&) followed by a *unary-expression*.

50 *addressof-expression:*

51 & *unary-expression*

52 Given an expression E which is of a type T and is classified as a fixed variable (§25.3), the construct &E computes the address of the variable given by E. The type of the result is T* and is classified as a value. A

1 compile-time error occurs if E is not classified as a variable, if E is classified as a volatile field (§17.4.3), or
 2 if E denotes a moveable variable. In the last case, a fixed statement (§25.6) can be used to temporarily “fix”
 3 the variable before obtaining its address.

4 The & operator does not require its argument to be definitely assigned, but following an & operation, the
 5 variable to which the operator is applied is considered definitely assigned in the execution path in which the
 6 operation occurs. It is the responsibility of the programmer to ensure that correct initialization of the variable
 7 actually does take place in this situation.

8 *[Example: In the example*

```

9     using System;
10    class Test
11    {
12        static void Main() {
13            int i;
14            unsafe {
15                int* p = &i;
16                *p = 123;
17            }
18            Console.WriteLine(i);
19        }
20    }

```

21 *i* is considered definitely assigned following the &*i* operation used to initialize *p*. The assignment to **p* in
 22 effect initializes *i*, but the inclusion of this initialization is the responsibility of the programmer, and no
 23 compile-time error would occur if the assignment was removed. *end example]*

24 *[Note: The rules of definite assignment for the & operator exist such that redundant initialization of local*
 25 *variables can be avoided. For example, many external APIs take a pointer to a structure which is filled in by*
 26 *the API. Calls to such APIs typically pass the address of a local struct variable, and without the rule,*
 27 *redundant initialization of the struct variable would be required. end note]*

28 *[Note: As stated in §14.5.4, outside an instance constructor or static constructor for a struct or class that*
 29 *defines a readonly field, that field is considered a value, not a variable. As such, its address cannot be taken.*
 30 *Similarly, the address of a constant cannot be taken. end note]*

31 25.5.5 Pointer increment and decrement

32 In an unsafe context, the ++ and -- operators (§14.5.9 and §14.6.5) can be applied to pointer variables of all
 33 types except void*. Thus, for every pointer type T*, the following operators are implicitly defined:

```

34     T* operator ++(T* x);
35     T* operator --(T* x);

```

36 The operators produce the same results as *x+1* and *x-1*, respectively (§25.5.6). In other words, for a pointer
 37 variable of type T*, the ++ operator adds sizeof(T) to the address contained in the variable, and the
 38 -- operator subtracts sizeof(T) from the address contained in the variable.

39 If a pointer increment or decrement operation overflows the domain of the pointer type, the result is
 40 implementation-defined, but no exceptions are produced.

41 25.5.6 Pointer arithmetic

42 In an unsafe context, the + operator (§14.7.4) and - operator (§14.7.5) can be applied to values of all
 43 pointer types except void*. Thus, for every pointer type T*, the following operators are implicitly defined:

```

44     T* operator +(T* x, int y);
45     T* operator +(T* x, uint y);
46     T* operator +(T* x, long y);
47     T* operator +(T* x, ulong y);
48     T* operator +(int x, T* y);
49     T* operator +(uint x, T* y);
50     T* operator +(long x, T* y);
51     T* operator +(ulong x, T* y);

```

```

1      T* operator -(T* x, int y);
2      T* operator -(T* x, uint y);
3      T* operator -(T* x, long y);
4      T* operator -(T* x, ulong y);
5      long operator -(T* x, T* y);

```

6 Given an expression *P* of a pointer type *T** and an expression *N* of type *int*, *uint*, *long*, or *ulong*, the
7 expressions *P* + *N* and *N* + *P* compute the pointer value of type *T** that results from adding
8 *N* * `sizeof(T)` to the address given by *P*. Likewise, the expression *P* - *N* computes the pointer value of
9 type *T** that results from subtracting *N* * `sizeof(T)` from the address given by *P*.

10 Given two expressions, *P* and *Q*, of a pointer type *T**, the expression *P* - *Q* computes the difference
11 between the addresses given by *P* and *Q* and then divides that difference by `sizeof(T)`. The type of the
12 result is always *long*. In effect, *P* - *Q* is computed as `((long)(P) - (long)(Q)) / sizeof(T)`.

13 [*Example*: For example:

```

14
15     using System;
16     class Test
17     {
18         static void Main() {
19             unsafe {
20                 int* values = stackalloc int[20];
21
22                 int* p = &values[1];
23                 int* q = &values[15];
24
25                 Console.WriteLine("p - q = {0}", p - q);
26                 Console.WriteLine("q - p = {0}", q - p);
27             }
28         }
29     }

```

30 which produces the output:

```

31     p - q = -14
32     q - p = 14

```

33 [*end example*]

34 If a pointer arithmetic operation overflows the domain of the pointer type, the result is truncated in an
35 implementation-defined fashion, but no exceptions are produced.

36 25.5.7 Pointer comparison

37 In an unsafe context, the `==`, `!=`, `<`, `>`, `<=`, and `=>` operators (§14.9) can be applied to values of all pointer
38 types. The pointer comparison operators are:

```

39     bool operator ==(void* x, void* y);
40     bool operator !=(void* x, void* y);
41     bool operator <(void* x, void* y);
42     bool operator >(void* x, void* y);
43     bool operator <=(void* x, void* y);
44     bool operator >=(void* x, void* y);

```

45 Because an implicit conversion exists from any pointer type to the `void*` type, operands of any pointer type
46 can be compared using these operators. The comparison operators compare the addresses given by the two
47 operands as if they were unsigned integers.

48 25.5.8 The `sizeof` operator

49 The `sizeof` operator returns the number of bytes occupied by a variable of a given type. The type specified
50 as an operand to `sizeof` must be an *unmanaged-type* (§25.2).

```

51     sizeof-expression:
52     sizeof ( unmanaged-type )

```

1 The result of the `sizeof` operator is a value of type `int`. For certain predefined types, the `sizeof` operator
 2 yields a constant value as shown in the table below.

3

Expression	Result
<code>sizeof(sbyte)</code>	1
<code>sizeof(byte)</code>	1
<code>sizeof(short)</code>	2
<code>sizeof(ushort)</code>	2
<code>sizeof(int)</code>	4
<code>sizeof(uint)</code>	4
<code>sizeof(long)</code>	8
<code>sizeof(ulong)</code>	8
<code>sizeof(char)</code>	2
<code>sizeof(float)</code>	4
<code>sizeof(double)</code>	8
<code>sizeof(bool)</code>	1

4

5 For all other types, the result of the `sizeof` operator is implementation-defined and is classified as a value,
 6 not a constant.

7 The order in which members are packed into a struct is unspecified.

8 For alignment purposes, there may be unnamed padding at the beginning of a struct, within a struct, and at
 9 the end of the struct. The contents of the bits used as padding are indeterminate.

10 When applied to an operand that has struct type, the result is the total number of bytes in a variable of that
 11 type, including any padding.

12 25.6 The `fixed` statement

13 In an unsafe context, the *embedded-statement* (§15) production permits an additional construct, the `fixed`
 14 statement, which is used to “fix” a moveable variable such that its address remains constant for the duration
 15 of the statement.

16 *embedded-statement*:

17 ...
 18 *fixed-statement*

19 *fixed-statement*:

20 `fixed (pointer-type fixed-pointer-declarators) embedded-statement`

21 *fixed-pointer-declarators*:

22 *fixed-pointer-declarator*
 23 *fixed-pointer-declarators* , *fixed-pointer-declarator*

24 *fixed-pointer-declarator*:

25 *identifier* = *fixed-pointer-initializer*

26

27 *fixed-pointer-initializer*:

28 `& variable-reference`
 29 *expression*

30 Each *fixed-pointer-declarator* declares a local variable of the given *pointer-type* and initializes that local
 31 variable with the address computed by the corresponding *fixed-pointer-initializer*. A local variable declared
 32 in a `fixed` statement is accessible in any *fixed-pointer-initializers* occurring to the right of that variable’s

1 declaration, and in the *embedded-statement* of the `fixed` statement. A local variable declared by a `fixed`
 2 statement is considered read-only. A compile-time error occurs if the embedded statement attempts to
 3 modify this local variable (via assignment or the `++` and `--` operators) or pass it as a `ref` or `out` parameter.

4 A *fixed-pointer-initializer* can be one of the following:

- 5 • The token “&” followed by a *variable-reference* (§12.3.3) to a moveable variable (§25.3) of an
 6 unmanaged type `T`, provided the type `T*` is implicitly convertible to the pointer type given in the `fixed`
 7 statement. In this case, the initializer computes the address of the given variable, and the variable is
 8 guaranteed to remain at a fixed address for the duration of the `fixed` statement.
- 9 • An expression of an *array-type* with elements of an unmanaged type `T`, provided the type `T*` is
 10 implicitly convertible to the pointer type given in the `fixed` statement. In this case, the initializer
 11 computes the address of the first element in the array, and the entire array is guaranteed to remain at a
 12 fixed address for the duration of the `fixed` statement. The behavior of the `fixed` statement is
 13 implementation-defined if the array expression is null or if the array has zero elements.
- 14 • An expression of type `string`, provided the type `char*` is implicitly convertible to the pointer type
 15 given in the `fixed` statement. In this case, the initializer computes the address of the first character in
 16 the string, and the entire string is guaranteed to remain at a fixed address for the duration of the `fixed`
 17 statement. The behavior of the `fixed` statement is implementation-defined if the string expression is
 18 null.

19 For each address computed by a *fixed-pointer-initializer* the `fixed` statement ensures that the variable
 20 referenced by the address is not subject to relocation or disposal by the garbage collector for the duration of
 21 the `fixed` statement. For example, if the address computed by a *fixed-pointer-initializer* references a field of
 22 an object or an element of an array instance, the `fixed` statement guarantees that the containing object
 23 instance is not relocated or disposed of during the lifetime of the statement.

24 It is the programmer's responsibility to ensure that pointers created by `fixed` statements do not survive
 25 beyond execution of those statements. For example, when pointers created by `fixed` statements are passed
 26 to external APIs, it is the programmer's responsibility to ensure that the APIs retain no memory of these
 27 pointers.

28 Fixed objects may cause fragmentation of the heap (because they can't be moved). For that reason, objects
 29 should be fixed only when absolutely necessary and then only for the shortest amount of time
 30 possible.[*Example:* The example

```

31     class Test
32     {
33         static int x;
34         int y;
35         unsafe static void F(int* p) {
36             *p = 1;
37         }
38         static void Main() {
39             Test t = new Test();
40             int[] a = new int[10];
41             unsafe {
42                 fixed (int* p = &x) F(p);
43                 fixed (int* p = &t.y) F(p);
44                 fixed (int* p = &a[0]) F(p);
45                 fixed (int* p = a) F(p);
46             }
47         }
48     }
  
```

49 demonstrates several uses of the `fixed` statement. The first statement fixes and obtains the address of a
 50 static field, the second statement fixes and obtains the address of an instance field, and the third statement
 51 fixes and obtains the address of an array element. In each case it would have been an error to use the regular
 52 `&` operator since the variables are all classified as moveable variables.

C# LANGUAGE SPECIFICATION

1 The third and fourth fixed statements in the example above produce identical results. In general, for an
2 array instance `a`, specifying `&a[0]` in a fixed statement is the same as simply specifying `a`.

3 Here's another example of the fixed statement, this time using `string`:

```
4
5     class Test
6     {
7         static string name = "xx";
8
9         unsafe static void F(char* p) {
10             for (int i = 0; p[i] != '\0'; ++i)
11                 Console.WriteLine(p[i]);
12         }
13
14         static void Main() {
15             unsafe {
16                 fixed (char* p = name) F(p);
17                 fixed (char* p = "xx") F(p);
18             }
19         }
20     }
```

21 *end example]*

22 In an unsafe context array elements of single-dimensional arrays are stored in increasing index order,
23 starting with index 0 and ending with index `Length - 1`. For multi-dimensional arrays, array elements are
24 stored such that the indices of the rightmost dimension are increased first, then the next left dimension, and
25 so on to the left.

26 Within a fixed statement that obtains a pointer `p` to an array instance `a`, the pointer values ranging from `p`
27 to `p + a.Length - 1` represent addresses of the elements in the array. Likewise, the variables ranging from
28 `p[0]` to `p[a.Length - 1]` represent the actual array elements. Given the way in which arrays are stored,
29 we can treat an array of any dimension as though it were linear. [*Example:* For example.

```
30     using System;
31     class Test
32     {
33         static void Main() {
34             int[, ,] a = new int[2,3,4];
35             unsafe {
36                 fixed (int* p = a) {
37                     for (int i = 0; i < a.Length; ++i) // treat as linear
38                         p[i] = i;
39                 }
40             }
41
42             for (int i = 0; i < 2; ++i)
43                 for (int j = 0; j < 3; ++j) {
44                     for (int k = 0; k < 4; ++k)
45                         console.write("[{0},{1},{2}] = {3,2} ", i, j, k,
46 a[i,j,k]);
47                     console.WriteLine();
48                 }
49             }
50     }
```

51 which produces the output:

```
52     [0,0,0] = 0 [0,0,1] = 1 [0,0,2] = 2 [0,0,3] = 3
53     [0,1,0] = 4 [0,1,1] = 5 [0,1,2] = 6 [0,1,3] = 7
54     [0,2,0] = 8 [0,2,1] = 9 [0,2,2] = 10 [0,2,3] = 11
55     [1,0,0] = 12 [1,0,1] = 13 [1,0,2] = 14 [1,0,3] = 15
56     [1,1,0] = 16 [1,1,1] = 17 [1,1,2] = 18 [1,1,3] = 19
57     [1,2,0] = 20 [1,2,1] = 21 [1,2,2] = 22 [1,2,3] = 23
```

58 *end example]*

59 [*Example:* In the example


```

1      class Test
2      {
3          unsafe static void Fill(int* p, int count, int value) {
4              for (; count != 0; count--) *p++ = value;
5          }
6          static void Main() {
7              int[] a = new int[100];
8              unsafe {
9                  fixed (int* p = a) Fill(p, 100, -1);
10             }
11         }
12     }

```

13 a *fixed* statement is used to fix an array so its address can be passed to a method that takes a pointer. *end example*

15 A `char*` value produced by fixing a string instance always points to a null-terminated string. Within a fixed statement that obtains a pointer `p` to a string instance `s`, the pointer values ranging from `p` to `p + s.Length - 1` represent addresses of the characters in the string, and the pointer value `p + s.Length` always points to a null character (the character with value `'\0'`).

19 Modifying objects of managed type through fixed pointers can result in undefined behavior. [*Note*: For example, because strings are immutable, it is the programmer's responsibility to ensure that the characters referenced by a pointer to a fixed string are not modified. *end note*]

22 [*Note*: The automatic null-termination of strings is particularly convenient when calling external APIs that expect "C-style" strings. Note, however, that a string instance is permitted to contain null characters. If such null characters are present, the string will appear truncated when treated as a null-terminated `char*`. *end note*]

26 25.7 Stack allocation

27 In an unsafe context, a local variable declaration (§15.5.1) may include a stack allocation initializer, which
28 allocates memory from the call stack.

```

29      local-variable-initializer:
30          expression
31          array-initializer
32          stackalloc-initializer

```

```

33      stackalloc-initializer:
34          stackalloc unmanaged-type [ expression ]

```

35 The *unmanaged-type* indicates the type of the items that will be stored in the newly allocated location, and
36 the *expression* indicates the number of these items. Taken together, these specify the required allocation
37 size. Since the size of a stack allocation cannot be negative, it is a compile-time error to specify the number
38 of items as a constant-expression that evaluates to a negative value.

39 A stack allocation initializer of the form `stackalloc T[E]` requires `T` to be an unmanaged type (§25.2) and
40 `E` to be an expression of type `int`. The construct allocates `E * sizeof(T)` bytes from the call stack and
41 returns a pointer, of type `T*`, to the newly allocated block. If `E` is a negative value, then the behavior is
42 undefined. If `E` is zero, then no allocation is made, and the pointer returned is implementation-defined. If
43 there is not enough memory available to allocate a block of the given size, a
44 `System.StackOverflowException` is thrown.

45 The content of the newly allocated memory is undefined.

46 Stack allocation initializers are not permitted in catch or finally blocks (§15.10).

47 [*Note*: There is no way to explicitly free memory allocated using `stackalloc`. *end note*] All stack-
48 allocated memory blocks created during the execution of a function member are automatically discarded
49 when that function member returns. [*Note*: This corresponds to the `alloca` function, an extension
50 commonly found C and C++ implementations. *end note*]

```

1  [Example: In the example
2      using System;
3      class Test
4      {
5          static string IntToString(int value) {
6              int n = value >= 0 ? value : -value;
7              unsafe {
8                  char* buffer = stackalloc char[16];
9                  char* p = buffer + 16;
10                 do {
11                     *--p = (char)(n % 10 + '0');
12                     n /= 10;
13                 } while (n != 0);
14                 if (value < 0) *--p = '-';
15                 return new string(p, 0, (int)(buffer + 16 - p));
16             }
17         }
18         static void Main() {
19             Console.WriteLine(IntToString(12345));
20             Console.WriteLine(IntToString(-999));
21         }
22     }

```

23 a `stackalloc` initializer is used in the `IntToString` method to allocate a buffer of 16 characters on the
24 stack. The buffer is automatically discarded when the method returns. *end example*]

25 25.8 Dynamic memory allocation

26 Except for the `stackalloc` operator, C# provides no predefined constructs for managing non-garbage
27 collected memory. Such services are typically provided by supporting class libraries or imported directly
28 from the underlying operating system. [Example: For example, the `Memory` class below illustrates how the
29 heap functions of an underlying operating system might be accessed from C#:

```

30     using System;
31     using System.Runtime.InteropServices;
32     public unsafe class Memory
33     {
34         // Handle for the process heap. This handle is used in all calls to
35         the // HeapXXX APIs in the methods below.
36         static int ph = GetProcessHeap();
37         // Private instance constructor to prevent instantiation.
38         private Memory() {}
39         // Allocates a memory block of the given size. The allocated memory is
40         // automatically initialized to zero.
41         public static void* Alloc(int size) {
42             void* result = HeapAlloc(ph, HEAP_ZERO_MEMORY, size);
43             if (result == null) throw new OutOfMemoryException();
44             return result;
45         }
46         // Copies count bytes from src to dst. The source and destination
47         // blocks are permitted to overlap.
48     }

```

```

1     public static void Copy(void* src, void* dst, int count) {
2         byte* ps = (byte*)src;
3         byte* pd = (byte*)dst;
4         if (ps > pd) {
5             for (; count != 0; count--) *pd++ = *ps++;
6         }
7         else if (ps < pd) {
8             for (ps += count, pd += count; count != 0; count--) *--pd = *--
9 ps;
10        }
11    }
12    // Frees a memory block.
13    public static void Free(void* block) {
14        if (!HeapFree(ph, 0, block)) throw new InvalidOperationException();
15    }
16    // Re-allocates a memory block. If the reallocation request is for a
17    // larger size, the additional region of memory is automatically
18    // initialized to zero.
19    public static void* ReAlloc(void* block, int size) {
20        void* result = HeapReAlloc(ph, HEAP_ZERO_MEMORY, block, size);
21        if (result == null) throw new OutOfMemoryException();
22        return result;
23    }
24    // Returns the size of a memory block.
25    public static int SizeOf(void* block) {
26        int result = HeapSize(ph, 0, block);
27        if (result == -1) throw new InvalidOperationException();
28        return result;
29    }
30    // Heap API flags
31    const int HEAP_ZERO_MEMORY = 0x00000008;
32    // Heap API functions
33    [DllImport("kernel32")]
34    static extern int GetProcessHeap();
35    [DllImport("kernel32")]
36    static extern void* HeapAlloc(int hHeap, int flags, int size);
37    [DllImport("kernel32")]
38    static extern bool HeapFree(int hHeap, int flags, void* block);
39    [DllImport("kernel32")]
40    static extern void* HeapReAlloc(int hHeap, int flags,
41        void* block, int size);
42    [DllImport("kernel32")]
43    static extern int HeapSize(int hHeap, int flags, void* block);
44 }

```

45 An example that uses the Memory class is given below:

```

46 class Test
47 {
48     static void Main() {
49         unsafe {
50             byte* buffer = (byte*)Memory.Alloc(256);
51             for (int i = 0; i < 256; i++) buffer[i] = (byte)i;
52             byte[] array = new byte[256];
53             fixed (byte* p = array) Memory.Copy(buffer, p, 256);
54             Memory.Free(buffer);
55             for (int i = 0; i < 256; i++) Console.WriteLine(array[i]);
56         }
57     }
58 }

```

C# LANGUAGE SPECIFICATION

1 The example allocates 256 bytes of memory through `Memory.Alloc` and initializes the memory block with
2 values increasing from 0 to 255. It then allocates a 256-element byte array and uses `Memory.Copy` to copy
3 the contents of the memory block into the byte array. Finally, the memory block is freed using
4 `Memory.Free` and the contents of the byte array are output on the console. *end example*]

5 **End of conditionally normative text.**

A. Grammar

1

2 **This clause is informative.**

3 This appendix contains summaries of the lexical and syntactic grammars found in the main document, and of
4 the grammar extensions for unsafe code. Grammar productions appear here in the same order that they
5 appear in the main document.

6 **A.1 Lexical grammar**

7 *input*::
8 *input-section*_{opt}
9 *input-section*::
10 *input-section-part*
11 *input-section input-section-part*
12 *input-section-part*::
13 *input-elements*_{opt} *new-line*
14 *pp-directive*
15 *input-elements*::
16 *input-element*
17 *input-elements input-element*
18 *input-element*::
19 *whitespace*
20 *comment*
21 *token*

22 **A.1.1 Line terminators**

23 *new-line*::
24 Carriage return character (U+000D)
25 Line feed character (U+000A)
26 Carriage return character (U+000D) followed by line feed character (U+000A)
27 Line separator character (U+2028)
28 Paragraph separator character (U+2029)

29 **A.1.2 White space**

30 *whitespace*::
31 Any character with Unicode class Zs
32 Horizontal tab character (U+0009)
33 Vertical tab character (U+000B)
34 Form feed character (U+000C)

35 **A.1.3 Comments**

36 *comment*::
37 *single-line-comment*
38 *delimited-comment*
39 *single-line-comment*::
40 *// input-characters*_{opt}

C# LANGUAGE SPECIFICATION

1 *input-characters*::
2 *input-character*
3 *input-characters* *input-character*

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

6 *new-line-character*::
7 Carriage return character (U+000D)
8 Line feed character (U+000A)
9 Line separator character (U+2028)
10 Paragraph separator character (U+2029)

11 *delimited-comment*::
12 */* delimited-comment-characters_{opt} */*

13 *delimited-comment-characters*::
14 *delimited-comment-character*
15 *delimited-comment-characters delimited-comment-character*

16 *delimited-comment-character*::
17 *not-asterisk*
18 ** not-slash*

19 *not-asterisk*::
20 Any Unicode character except ***

21 *not-slash*::
22 Any Unicode character except */*

23 **A.1.4 Tokens**

24 *token*::
25 *identifier*
26 *keyword*
27 *integer-literal*
28 *real-literal*
29 *character-literal*
30 *string-literal*
31 *operator-or-punctuator*

32 **A.1.5 Unicode character escape sequences**

33 *unicode-character-escape-sequence*::
34 *\u hex-digit hex-digit hex-digit hex-digit*
35 *\U hex-digit hex-digit hex-digit hex-digit hex-digit hex-digit hex-digit hex-digit*

36 **A.1.6 Identifiers**

37 *identifier*::
38 *available-identifier*
39 *@ identifier-or-keyword*

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

42 *identifier-or-keyword*::
43 *identifier-start-character identifier-part-characters_{opt}*

44 *identifier-start-character*::
45 *letter-character*
46 *_* (the underscore character)

1 *identifier-part-characters::*
2 *identifier-part-character*
3 *identifier-part-characters identifier-part-character*

4 *identifier-part-character::*
5 *letter-character*
6 *decimal-digit-character*
7 *connecting-character*
8 *combining-character*
9 *formatting-character*

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

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

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

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

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

26 **A.1.7 Keywords**

27 *keyword::* one of

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

1 **A.1.8 Literals**2 *literal::*3 *boolean-literal*4 *integer-literal*5 *real-literal*6 *character-literal*7 *string-literal*8 *null-literal*9 *boolean-literal::*10 **true**11 **false**12 *integer-literal::*13 *decimal-integer-literal*14 *hexadecimal-integer-literal*15 *decimal-integer-literal::*16 *decimal-digits integer-type-suffix_{opt}*17 *decimal-digits::*18 *decimal-digit*19 *decimal-digits decimal-digit*20 *decimal-digit:: one of*21 **0 1 2 3 4 5 6 7 8 9**22 *integer-type-suffix:: one of*23 **U u L l UL Ul uL ul LU Lu lU lu**24 *hexadecimal-integer-literal::*25 **0x** *hex-digits integer-type-suffix_{opt}*26 **0X** *hex-digits integer-type-suffix_{opt}*27 *hex-digits::*28 *hex-digit*29 *hex-digits hex-digit*30 *hex-digit:: one of*31 **0 1 2 3 4 5 6 7 8 9 A B C D E F a b c d e f**32 *real-literal::*33 *decimal-digits . decimal-digits exponent-part_{opt} real-type-suffix_{opt}*34 *. decimal-digits exponent-part_{opt} real-type-suffix_{opt}*35 *decimal-digits exponent-part real-type-suffix_{opt}*36 *decimal-digits real-type-suffix*37 *exponent-part::*38 **e** *sign_{opt} decimal-digits*39 **E** *sign_{opt} decimal-digits*40 *sign:: one of*41 **+ -**42 *real-type-suffix:: one of*43 **F f D d M m**44 *character-literal::*45 **' character '**


```

1  character::
2      single-character
3      simple-escape-sequence
4      hexadecimal-escape-sequence
5      unicode-character-escape-sequence
6
7  single-character::
8      Any character except ' (U+0027), \ (U+005C), and new-line-character
9
10 simple-escape-sequence:: one of
11     \' \" \\ \0 \a \b \f \n \r \t \v
12
13 hexadecimal-escape-sequence::
14     \x hex-digit hex-digitopt hex-digitopt hex-digitopt
15
16 string-literal::
17     regular-string-literal
18     verbatim-string-literal
19
20 regular-string-literal::
21     " regular-string-literal-charactersopt "
22
23 regular-string-literal-characters::
24     regular-string-literal-character
25     regular-string-literal-characters regular-string-literal-character
26
27 regular-string-literal-character::
28     single-regular-string-literal-character
29     simple-escape-sequence
30     hexadecimal-escape-sequence
31     unicode-character-escape-sequence
32
33 single-regular-string-literal-character::
34     Any character except " (U+0022), \ (U+005C), and new-line-character
35
36 verbatim-string-literal::
37     @" verbatim-string-literal-charactersopt "
38
39 verbatim-string-literal-characters::
40     verbatim-string-literal-character
41     verbatim-string-literal-characters verbatim-string-literal-character
42
43 verbatim-string-literal-character::
44     single-verbatim-string-literal-character
45     quote-escape-sequence
46
47 single-verbatim-string-literal-character::
48     any character except "
49
50 quote-escape-sequence::
51     ""
52
53 null-literal::
54     null

```

41 A.1.9 Operators and punctuators

```

42 operator-or-punctuator:: one of
43 {      }      [      ]      (      )      .      ,      :      ;
44 +      -      *      /      %      &      |      ^      !      ~
45 =      <      >      ?      ++     --     &&     ||     <<     >>
46 ==     !=     <=     >=     +=     -=     *=     /=     %=     &=
47 |=     ^=     <<=     >>=     ->

```

A.1.10 Pre-processing directives

```

1  pp-directive::
2      pp-declaration
3      pp-conditional
4      pp-line
5      pp-diagnostic
6      pp-region
7
8  pp-new-line::
9      whitespaceopt single-line-commentopt new-line
10
11 conditional-symbol::
12     Any identifier-or-keyword except true or false
13
14 pp-expression::
15     whitespaceopt pp-or-expression whitespaceopt
16
17 pp-or-expression::
18     pp-and-expression
19     pp-or-expression whitespaceopt || whitespaceopt pp-and-expression
20
21 pp-and-expression::
22     pp-equality-expression
23     pp-and-expression whitespaceopt && whitespaceopt pp-equality-expression
24
25 pp-equality-expression::
26     pp-unary-expression
27     pp-equality-expression whitespaceopt == whitespaceopt pp-unary-expression
28     pp-equality-expression whitespaceopt != whitespaceopt pp-unary-expression
29
30 pp-unary-expression::
31     pp-primary-expression
32     ! whitespaceopt pp-unary-expression
33
34 pp-primary-expression::
35     true
36     false
37     conditional-symbol
38     ( whitespaceopt pp-expression whitespaceopt )
39
40
41 pp-declaration::
42     whitespaceopt # whitespaceopt define whitespace conditional-symbol pp-new-line
43     whitespaceopt # whitespaceopt undef whitespace conditional-symbol pp-new-line
44
45 pp-conditional::
46     pp-if-section pp-elif-sectionsopt pp-else-sectionopt pp-endif
47
48 pp-if-section::
49     whitespaceopt # whitespaceopt if whitespace pp-expression pp-new-line conditional-
50     sectionopt
51
52 pp-elif-sections::
53     pp-elif-section
54     pp-elif-sections pp-elif-section
55
56 pp-elif-section::
57     whitespaceopt # whitespaceopt elif whitespace pp-expression pp-new-line conditional-
58     sectionopt
59
60 pp-else-section::
61     whitespaceopt # whitespaceopt else pp-new-line conditional-sectionopt

```

```

1  pp-endif::
2      whitespaceopt # whitespaceopt endif pp-new-line
3
4  conditional-section::
5      input-section
6      skipped-section
7
8  skipped-section::
9      skipped-section-part
10     skipped-section skipped-section-part
11
12  skipped-section-part::
13     skipped-charactersopt new-line
14     pp-directive
15
16  skipped-characters::
17     whitespaceopt not-number-sign input-charactersopt
18
19  not-number-sign::
20     Any input-character except #
21
22  pp-line::
23     whitespaceopt # whitespaceopt line whitespaceopt line-indicator pp-new-line
24
25  line-indicator::
26     decimal-digits whitespace file-name
27     decimal-digits
28     default
29
30  file-name::
31     " file-name-characters "
32
33  file-name-characters::
34     file-name-character
35     file-name-characters file-name-character
36
37  file-name-character::
38     Any character except " (U+0022), and new-line
39
40  pp-diagnostic::
41     whitespaceopt # whitespaceopt error whitespaceopt pp-message
42     whitespaceopt # whitespaceopt warning whitespaceopt pp-message
43
44  pp-message::
45     input-charactersopt new-line
46
47  pp-region::
48     pp-start-region conditional-sectionopt pp-end-region
49
50  pp-start-region::
51     whitespaceopt # whitespaceopt region whitespaceopt pp-message
52
53  pp-end-region::
54     whitespaceopt # whitespaceopt endregion whitespaceopt pp-message

```

40 A.2 Syntactic grammar

41 A.2.1 Basic concepts

```

42     namespace-name:
43         namespace-or-type-name
44
45     type-name:
46         namespace-or-type-name

```

1 *namespace-or-type-name*:
2 *identifier*
3 *namespace-or-type-name* . *identifier*

4 **A.2.2 Types**

5 *type*:
6 *value-type*
7 *reference-type*

8 *value-type*:
9 *struct-type*
10 *enum-type*

11 *struct-type*:
12 *type-name*
13 *simple-type*

14 *simple-type*:
15 *numeric-type*
16 **bool**

17 *numeric-type*:
18 *integral-type*
19 *floating-point-type*
20 **decimal**

21 *integral-type*:
22 **sbyte**
23 **byte**
24 **short**
25 **ushort**
26 **int**
27 **uint**
28 **long**
29 **ulong**
30 **char**

31 *floating-point-type*:
32 **float**
33 **double**

34 *enum-type*:
35 *type-name*

36 *reference-type*:
37 *class-type*
38 *interface-type*
39 *array-type*
40 *delegate-type*

41 *class-type*:
42 *type-name*
43 **object**
44 **string**

45 *interface-type*:
46 *type-name*

47 *array-type*:
48 *non-array-type* *rank-specifiers*

1 *non-array-type:*
 2 *type*

3 *rank-specifiers:*
 4 *rank-specifier*
 5 *rank-specifiers rank-specifier*

6 *rank-specifier:*
 7 [*dim-separators*_{opt}]

8 *dim-separators:*
 9 ,
 10 *dim-separators* ,

11 *delegate-type:*
 12 *type-name*

13 **A.2.3 Variables**

14 *variable-reference:*
 15 *expression*

16 **A.2.4 Expressions**

17 *argument-list:*
 18 *argument*
 19 *argument-list* , *argument*

20 *argument:*
 21 *expression*
 22 **ref** *variable-reference*
 23 **out** *variable-reference*

24 *primary-expression:*
 25 *array-creation-expression*
 26 *primary-no-array-creation-expression*

27 *primary-no-array-creation-expression:*
 28 *literal*
 29 *simple-name*
 30 *parenthesized-expression*
 31 *member-access*
 32 *invocation-expression*
 33 *element-access*
 34 *this-access*
 35 *base-access*
 36 *post-increment-expression*
 37 *post-decrement-expression*
 38 *object-creation-expression*
 39 *delegate-creation-expression*

40

41 *typeof-expression*
 42 *sizeof-expression*
 43 *checked-expression*
 44 *unchecked-expression*

45 *simple-name:*
 46 *identifier*

47 *parenthesized-expression:*
 48 (*expression*)

C# LANGUAGE SPECIFICATION

1 *member-access*:
2 *primary-expression* . *identifier*
3 *predefined-type* . *identifier*

4 *predefined-type*: one of
5 *bool* *byte* *char* *decimal* *double* *float* *int* *long*
6 *object* *sbyte* *short* *string* *uint* *ulong* *ushort*

7 *invocation-expression*:
8 *primary-expression* (*argument-list*_{opt})

9 *element-access*:
10 *primary-no-array-creation-expression* [*expression-list*]

11 *expression-list*:
12 *expression*
13 *expression-list* , *expression*

14 *this-access*:
15 *this*

16 *base-access*:
17 *base* . *identifier*
18 *base* [*expression-list*]

19 *post-increment-expression*:
20 *primary-expression* ++

21 *post-decrement-expression*:
22 *primary-expression* --

23 *object-creation-expression*:
24 *new type* (*argument-list*_{opt})

25 *array-creation-expression*:
26 *new non-array-type* [*expression-list*] *rank-specifiers*_{opt} *array-initializer*_{opt}
27 *new array-type* *array-initializer*

28 *delegate-creation-expression*:
29 *new delegate-type* (*expression*)

30 *typeof-expression*:
31 *typeof* (*type*)
32 *typeof* (*void*)

33 *checked-expression*:
34 *checked* (*expression*)

35 *unchecked-expression*:
36 *unchecked* (*expression*)

37 *unary-expression*:
38 *primary-expression*
39 + *unary-expression*
40 - *unary-expression*
41 ! *unary-expression*
42 ~ *unary-expression*
43 * *unary-expression*
44 *pre-increment-expression*
45 *pre-decrement-expression*
46 *cast-expression*

47 *pre-increment-expression*:
48 ++ *unary-expression*

1 *pre-decrement-expression:*
2 -- *unary-expression*

3 *cast-expression:*
4 (*type*) *unary-expression*

5 *multiplicative-expression:*
6 *unary-expression*
7 *multiplicative-expression* * *unary-expression*
8 *multiplicative-expression* / *unary-expression*
9 *multiplicative-expression* % *unary-expression*

10 *additive-expression:*
11 *multiplicative-expression*
12 *additive-expression* + *multiplicative-expression*
13 *additive-expression* - *multiplicative-expression*

14 *shift-expression:*
15 *additive-expression*
16 *shift-expression* << *additive-expression*
17 *shift-expression* >> *additive-expression*

18 *relational-expression:*
19 *shift-expression*
20 *relational-expression* < *shift-expression*
21 *relational-expression* > *shift-expression*
22 *relational-expression* <= *shift-expression*
23 *relational-expression* >= *shift-expression*
24 *relational-expression* **is** *type*
25 *relational-expression* **as** *type*

26 *equality-expression:*
27 *relational-expression*
28 *equality-expression* == *relational-expression*
29 *equality-expression* != *relational-expression*

30 *and-expression:*
31 *equality-expression*
32 *and-expression* & *equality-expression*

33 *exclusive-or-expression:*
34 *and-expression*
35 *exclusive-or-expression* ^ *and-expression*

36 *inclusive-or-expression:*
37 *exclusive-or-expression*
38 *inclusive-or-expression* | *exclusive-or-expression*

39 *conditional-and-expression:*
40 *inclusive-or-expression*
41 *conditional-and-expression* && *inclusive-or-expression*

42 *conditional-or-expression:*
43 *conditional-and-expression*
44 *conditional-or-expression* || *conditional-and-expression*

45 *conditional-expression:*
46 *conditional-or-expression*
47 *conditional-or-expression* ? *expression* : *expression*

48 *assignment:*
49 *unary-expression* *assignment-operator* *expression*

C# LANGUAGE SPECIFICATION

1 *assignment-operator*: one of
2 = += -= *= /= %= &= |= ^= <<= >>=
3 *expression*:
4 *conditional-expression*
5 *assignment*
6 *constant-expression*:
7 *expression*
8 *boolean-expression*:
9 *expression*

10 **A.2.5 Statements**

11 *statement*:
12 *labeled-statement*
13 *declaration-statement*
14 *embedded-statement*
15 *embedded-statement*:
16 *block*
17 *empty-statement*
18 *expression-statement*
19 *selection-statement*
20 *iteration-statement*
21 *jump-statement*
22 *try-statement*
23 *checked-statement*
24 *unchecked-statement*
25 *lock-statement*
26 *using-statement*
27 *block*:
28 { *statement-list*_{opt} }
29 *statement-list*:
30 *statement*
31 *statement-list* *statement*
32 *empty-statement*:
33 ;
34 *labeled-statement*:
35 *identifier* : *statement*
36 *declaration-statement*:
37 *local-variable-declaration* ;
38 *local-constant-declaration* ;
39 *local-variable-declaration*:
40 *type* *local-variable-declarators*
41 *local-variable-declarators*:
42 *local-variable-declarator*
43 *local-variable-declarators* , *local-variable-declarator*
44 *local-variable-declarator*:
45 *identifier*
46 *identifier* = *local-variable-initializer*

1 *local-variable-initializer:*
2 *expression*
3 *array-initializer*

4 *local-constant-declaration:*
5 **const** *type* *constant-declarators*

6 *constant-declarators:*
7 *constant-declarator*
8 *constant-declarators* , *constant-declarator*

9 *constant-declarator:*
10 *identifier* = *constant-expression*

11 *expression-statement:*
12 *statement-expression* ;

13 *statement-expression:*
14 *invocation-expression*
15 *object-creation-expression*
16 *assignment*
17 *post-increment-expression*
18 *post-decrement-expression*
19 *pre-increment-expression*
20 *pre-decrement-expression*

21 *selection-statement:*
22 *if-statement*
23 *switch-statement*

24 *if-statement:*
25 **if** (*boolean-expression*) *embedded-statement*
26 **if** (*boolean-expression*) *embedded-statement* **else** *embedded-statement*

27 *boolean-expression:*
28 *expression*

29 *switch-statement:*
30 **switch** (*expression*) *switch-block*

31 *switch-block:*
32 { *switch-sections*_{opt} }

33 *switch-sections:*
34 *switch-section*
35 *switch-sections* *switch-section*

36 *switch-section:*
37 *switch-labels* *statement-list*

38 *switch-labels:*
39 *switch-label*
40 *switch-labels* *switch-label*

41 *switch-label:*
42 **case** *constant-expression* :
43 **default** :

44 *iteration-statement:*
45 *while-statement*
46 *do-statement*
47 *for-statement*
48 *foreach-statement*

C# LANGUAGE SPECIFICATION

1 *while-statement:*
2 *while* (*boolean-expression*) *embedded-statement*

3 *do-statement:*
4 *do* *embedded-statement* *while* (*boolean-expression*) ;

5 *for-statement:*
6 *for* (*for-initializer*_{opt} ; *for-condition*_{opt} ; *for-iterator*_{opt}) *embedded-statement*

7 *for-initializer:*
8 *local-variable-declaration*
9 *statement-expression-list*

10 *for-condition:*
11 *boolean-expression*

12 *for-iterator:*
13 *statement-expression-list*

14 *statement-expression-list:*
15 *statement-expression*
16 *statement-expression-list* , *statement-expression*

17 *foreach-statement:*
18 *foreach* (*type identifier in expression*) *embedded-statement*

19 *jump-statement:*
20 *break-statement*
21 *continue-statement*
22 *goto-statement*
23 *return-statement*
24 *throw-statement*

25 *break-statement:*
26 *break* ;

27 *continue-statement:*
28 *continue* ;

29 *goto-statement:*
30 *goto* *identifier* ;
31 *goto* *case constant-expression* ;
32 *goto* *default* ;

33 *return-statement:*
34 *return* *expression*_{opt} ;

35 *throw-statement:*
36 *throw* *expression*_{opt} ;

37 *try-statement:*
38 *try* *block* *catch-clauses*
39 *try* *block* *finally-clause*
40 *try* *block* *catch-clauses* *finally-clause*

41 *catch-clauses:*
42 *specific-catch-clauses* *general-catch-clause*_{opt}
43 *specific-catch-clauses*_{opt} *general-catch-clause*

44 *specific-catch-clauses:*
45 *specific-catch-clause*
46 *specific-catch-clauses* *specific-catch-clause*

```

1  specific-catch-clause:
2      catch ( class-type identifieropt ) block
3
4  general-catch-clause:
5      catch block
6
7  finally-clause:
8      finally block
9
10 checked-statement:
11     checked block
12
13 unchecked-statement:
14     unchecked block
15
16 lock-statement:
17     lock ( expression ) embedded-statement
18
19 using-statement:
20     using ( resource-acquisition ) embedded-statement
21
22 resource-acquisition:
23     local-variable-declaration
24     expression
25
26 compilation-unit:
27     using-directivesopt global-attributesopt namespace-member-declarationsopt
28
29 namespace-declaration:
30     namespace qualified-identifier namespace-body ;opt
31
32 qualified-identifier:
33     identifier
34     qualified-identifier . identifier
35
36 namespace-body:
37     { using-directivesopt namespace-member-declarationsopt }
38
39 using-directives:
40     using-directive
41     using-directives using-directive
42
43 using-directive:
44     using-alias-directive
45     using-namespace-directive
46
47 using-alias-directive:
48     using identifier = namespace-or-type-name ;
49
50 using-namespace-directive:
51     using namespace-name ;
52
53 namespace-member-declarations:
54     namespace-member-declaration
55     namespace-member-declarations namespace-member-declaration
56
57 namespace-member-declaration:
58     namespace-declaration
59     type-declaration

```

1 *type-declaration:*
 2 *class-declaration*
 3 *struct-declaration*
 4 *interface-declaration*
 5 *enum-declaration*
 6 *delegate-declaration*

7 **A.2.6 Classes**

8 *class-declaration:*
 9 *attributes*_{opt} *class-modifiers*_{opt} **class** *identifier* *class-base*_{opt} *class-body* ;_{opt}

10 *class-modifiers:*
 11 *class-modifier*
 12 *class-modifiers* *class-modifier*

13 *class-modifier:*
 14 **new**
 15 **public**
 16 **protected**
 17 **internal**
 18 **private**
 19 **abstract**
 20 **sealed**

21 *class-base:*
 22 : *class-type*
 23 : *interface-type-list*
 24 : *class-type* , *interface-type-list*

25 *interface-type-list:*
 26 *interface-type*
 27 *interface-type-list* , *interface-type*

28 *class-body:*
 29 { *class-member-declarations*_{opt} }

30 *class-member-declarations:*
 31 *class-member-declaration*
 32 *class-member-declarations* *class-member-declaration*

33 *class-member-declaration:*
 34 *constant-declaration*
 35 *field-declaration*
 36 *method-declaration*
 37 *property-declaration*
 38 *event-declaration*
 39 *indexer-declaration*
 40 *operator-declaration*
 41 *constructor-declaration*
 42 *destructor-declaration*
 43 *static-constructor-declaration*
 44 *type-declaration*

45 *constant-declaration:*
 46 *attributes*_{opt} *constant-modifiers*_{opt} **const** *type* *constant-declarators* ;

47 *constant-modifiers:*
 48 *constant-modifier*
 49 *constant-modifiers* *constant-modifier*

```

1      constant-modifier:
2          new
3          public
4          protected
5          internal
6          private
7
8      constant-declarators:
9          constant-declarator
10         constant-declarators , constant-declarator
11
12     constant-declarator:
13         identifier = constant-expression
14
15     field-declaration:
16         attributesopt field-modifiersopt type variable-declarators ;
17
18     field-modifiers:
19         field-modifier
20         field-modifiers field-modifier
21
22     field-modifier:
23         new
24         public
25         protected
26         internal
27         private
28         static
29         readonly
30         volatile
31
32     variable-declarators:
33         variable-declarator
34         variable-declarators , variable-declarator
35
36     variable-declarator:
37         identifier
38         identifier = variable-initializer
39
40     variable-initializer:
41         expression
42         array-initializer
43
44     method-declaration:
45         method-header method-body
46
47     method-header:
48         attributesopt method-modifiersopt return-type member-name ( formal-parameter-listopt )
49
50     method-modifiers:
51         method-modifier
52         method-modifiers method-modifier

```

C# LANGUAGE SPECIFICATION

1 *method-modifier:*
2 *new*
3 *public*
4 *protected*
5 *internal*
6 *private*
7 *static*
8 *virtual*
9 *sealed*
10 *override*
11 *abstract*
12 *extern*

13 *return-type:*
14 *type*
15 *void*

16 *member-name:*
17 *identifier*
18 *interface-type . identifier*

19 *method-body:*
20 *block*
21 *;*

22 *formal-parameter-list:*
23 *fixed-parameters*
24 *fixed-parameters , parameter-array*
25 *parameter-array*

26 *fixed-parameters:*
27 *fixed-parameter*
28 *fixed-parameters , fixed-parameter*

29 *fixed-parameter:*
30 *attributes_{opt} parameter-modifier_{opt} type identifier*

31 *parameter-modifier:*
32 *ref*
33 *out*

34 *parameter-array:*
35 *attributes_{opt} params array-type identifier*

36 *property-declaration:*
37 *attributes_{opt} property-modifiers_{opt} type member-name { accessor-declarations }*

38 *property-modifiers:*
39 *property-modifier*
40 *property-modifiers property-modifier*

```

1  property-modifier:
2      new
3      public
4      protected
5      internal
6      private
7      static
8      virtual
9      sealed
10     override
11     abstract
12     extern
13
14 member-name:
15     identifier
16     interface-type . identifier
17
18 accessor-declarations:
19     get-accessor-declaration set-accessor-declarationopt
20     set-accessor-declaration get-accessor-declarationopt
21
22 get-accessor-declaration:
23     attributesopt get accessor-body
24
25 set-accessor-declaration:
26     attributesopt set accessor-body
27
28 accessor-body:
29     block
30     ;
31
32 event-declaration:
33     attributesopt event-modifiersopt event type variable-declarators ;
34     attributesopt event-modifiersopt event type member-name { event-accessor-declarations
35     }
36
37 event-modifiers:
38     event-modifier
39     event-modifiers event-modifier
40
41 event-modifier:
42     new
43     public
44     protected
45     internal
46     private
47     static
48     virtual
49     sealed
50     override
51     abstract
52     extern
53
54 event-accessor-declarations:
55     add-accessor-declaration remove-accessor-declaration
56     remove-accessor-declaration add-accessor-declaration
57
58 add-accessor-declaration:
59     attributesopt add block

```

C# LANGUAGE SPECIFICATION

```
1      remove-accessor-declaration:
2          attributesopt remove block
3
4      indexer-declaration:
5          attributesopt indexer-modifiersopt indexer-declarator { accessor-declarations }
6
7      indexer-modifiers:
8          indexer-modifier
9          indexer-modifiers indexer-modifier
10
11     indexer-modifier:
12         new
13         public
14         protected
15         internal
16         private
17         virtual
18         sealed
19         override
20         abstract
21         extern
22
23     indexer-declarator:
24         type this [ formal-parameter-list ]
25         type interface-type . this [ formal-parameter-list ]
26
27     operator-declaration:
28         attributesopt operator-modifiers operator-declarator operator-body
29
30     operator-modifiers:
31         operator-modifier
32         operator-modifiers operator-modifier
33
34     operator-modifier:
35         public
36         static
37         extern
38
39     operator-declarator:
40         unary-operator-declarator
41         binary-operator-declarator
42         conversion-operator-declarator
43
44     unary-operator-declarator:
45         type operator overloadable-unary-operator ( type identifier )
46
47     overloadable-unary-operator: one of
48         + - ! ~ ++ -- true false
49
50     binary-operator-declarator:
51         type operator overloadable-binary-operator ( type identifier , type identifier )
52
53     overloadable-binary-operator: one of
54         + - * / % & | ^ << >> == != > < >= <=
55
56     conversion-operator-declarator:
57         implicit operator type ( type identifier )
58         explicit operator type ( type identifier )
59
60     operator-body:
61         block
62         ;
```


1 *constructor-declaration:*
2 *attributes_{opt} constructor-modifiers_{opt} constructor-declarator constructor-body*

3 *constructor-modifiers:*
4 *constructor-modifier*
5 *constructor-modifiers constructor-modifier*

6 *constructor-modifier:*
7 *public*
8 *protected*
9 *internal*
10 *private*
11 *extern*

12 *constructor-declarator:*
13 *identifier (formal-parameter-list_{opt}) constructor-initializer_{opt}*

14 *constructor-initializer:*
15 *: base (argument-list_{opt})*
16 *: this (argument-list_{opt})*

17 *constructor-body:*
18 *block*
19 *;*

20 *static-constructor-declaration:*
21 *attributes_{opt} static-constructor-modifiers identifier () static-constructor-body*

22 *static-constructor-modifiers:*
23 *extern_{opt} static*
24 *static extern_{opt}*

25 *static-constructor-body:*
26 *block*
27 *;*

28 *destructor-declaration:*
29 *attributes_{opt} extern_{opt} ~ identifier () destructor-body*

30 *destructor-body:*
31 *block*
32 *;*

33 **A.2.7 Structs**

34 *struct-declaration:*
35 *attributes_{opt} struct-modifiers_{opt} struct identifier struct-interfaces_{opt} struct-body ;_{opt}*

36 *struct-modifiers:*
37 *struct-modifier*
38 *struct-modifiers struct-modifier*

39 *struct-modifier:*
40 *new*
41 *public*
42 *protected*
43 *internal*
44 *private*

45 *struct-interfaces:*
46 *: interface-type-list*

1 *struct-body*:
 2 { *struct-member-declarations*_{opt} }

3 *struct-member-declarations*:
 4 *struct-member-declaration*
 5 *struct-member-declarations struct-member-declaration*

6 *struct-member-declaration*:
 7 *constant-declaration*
 8 *field-declaration*
 9 *method-declaration*
 10 *property-declaration*
 11 *event-declaration*
 12 *indexer-declaration*
 13 *operator-declaration*
 14 *constructor-declaration*
 15 *static-constructor-declaration*
 16 *type-declaration*

17 **A.2.8 Arrays**

18 *array-type*:
 19 *non-array-type rank-specifiers*

20 *non-array-type*:
 21 *type*

22 *rank-specifiers*:
 23 *rank-specifier*
 24 *rank-specifiers rank-specifier*

25 *rank-specifier*:
 26 [*dim-separators*_{opt}]

27 *dim-separators*:
 28 ,
 29 *dim-separators* ,

30 *array-initializer*:
 31 { *variable-initializer-list*_{opt} }
 32 { *variable-initializer-list* , }

33 *variable-initializer-list*:
 34 *variable-initializer*
 35 *variable-initializer-list* , *variable-initializer*

36 *variable-initializer*:
 37 *expression*
 38 *array-initializer*

39 **A.2.9 Interfaces**

40 *interface-declaration*:
 41 *attributes*_{opt} *interface-modifiers*_{opt} **interface** *identifier* *interface-base*_{opt} *interface-body*
 42 ;_{opt}

43 *interface-modifiers*:
 44 *interface-modifier*
 45 *interface-modifiers interface-modifier*

```

1  interface-modifier:
2      new
3      public
4      protected
5      internal
6      private
7
8  interface-base:
9      : interface-type-list
10
11 interface-body:
12     { interface-member-declarationsopt }
13
14 interface-member-declarations:
15     interface-member-declaration
16     interface-member-declarations interface-member-declaration
17
18 interface-member-declaration:
19     interface-method-declaration
20     interface-property-declaration
21     interface-event-declaration
22     interface-indexer-declaration
23
24 interface-method-declaration:
25     attributesopt newopt return-type identifier ( formal-parameter-listopt ) ;
26
27 interface-property-declaration:
28     attributesopt newopt type identifier { interface-accessors }
29
30 interface-accessors:
31     attributesopt get ;
32     attributesopt set ;
33     attributesopt get ; attributesopt set ;
34     attributesopt set ; attributesopt get ;
35
36 interface-event-declaration:
37     attributesopt newopt event type identifier ;
38
39 interface-indexer-declaration:
40     attributesopt newopt type this [ formal-parameter-list ] { interface-accessors }

```

A.2.10 Enums

```

41 enum-declaration:
42     attributesopt enum-modifiersopt enum identifier enum-baseopt enum-body ;opt
43
44 enum-base:
45     : integral-type
46
47 enum-body:
48     { enum-member-declarationsopt }
49     { enum-member-declarations , }
50
51 enum-modifiers:
52     enum-modifier
53     enum-modifiers enum-modifier
54
55 enum-modifier:
56     new
57     public
58     protected
59     internal
60     private

```

1 *enum-member-declarations:*
 2 *enum-member-declaration*
 3 *enum-member-declarations* , *enum-member-declaration*
 4 *enum-member-declaration:*
 5 *attributes*_{opt} *identifier*
 6 *attributes*_{opt} *identifier* = *constant-expression*

A.2.11 Delegates

8 *delegate-declaration:*
 9 *attributes*_{opt} *delegate-modifiers*_{opt} **delegate** *type* *identifier* (*formal-parameter-list*_{opt})
 10 ;
 11 *delegate-modifiers:*
 12 *delegate-modifier*
 13 *delegate-modifiers* *delegate-modifier*
 14 *delegate-modifier:*
 15 **new**
 16 **public**
 17 **protected**
 18 **internal**
 19 **private**

A.2.12 Attributes

21 *global-attributes:*
 22 *global-attribute-sections*
 23 *global-attribute-sections:*
 24 *global-attribute-section*
 25 *global-attribute-sections* *global-attribute-section*
 26 *global-attribute-section:*
 27 [*global-attribute-target-specifier* *attribute-list*]
 28 [*global-attribute-target-specifier* *attribute-list* ,]
 29 *global-attribute-target-specifier:*
 30 *global-attribute-target* ;
 31 *global-attribute-target:*
 32 **assembly**
 33 *attributes:*
 34 *attribute-sections*
 35 *attribute-sections:*
 36 *attribute-section*
 37 *attribute-sections* *attribute-section*
 38 *attribute-section:*
 39 [*attribute-target-specifier*_{opt} *attribute-list*]
 40 [*attribute-target-specifier*_{opt} *attribute-list* ,]
 41 *attribute-target-specifier:*
 42 *attribute-target* ;

1 *attribute-target:*
 2 field
 3 event
 4 method
 5 module
 6 param
 7 property
 8 return
 9 type
 10 *attribute-list:*
 11 *attribute*
 12 *attribute-list* , *attribute*
 13 *attribute:*
 14 *attribute-name* *attribute-arguments*_{opt}
 15 *attribute-name:*
 16 *type-name*
 17 *attribute-arguments:*
 18 (*positional-argument-list*_{opt})
 19 (*positional-argument-list* , *named-argument-list*)
 20 (*named-argument-list*)
 21 *positional-argument-list:*
 22 *positional-argument*
 23 *positional-argument-list* , *positional-argument*
 24 *positional-argument:*
 25 *attribute-argument-expression*
 26 *named-argument-list:*
 27 *named-argument*
 28 *named-argument-list* , *named-argument*
 29 *named-argument:*
 30 *identifier* = *attribute-argument-expression*
 31 *attribute-argument-expression:*
 32 *expression*

33 **A.3 Grammar extensions for unsafe code**

34 *embedded-statement:*
 35 ...
 36 *unsafe-statement*
 37 *unsafe-statement:*
 38 unsafe *block*
 39 *type:*
 40 *value-type*
 41 *reference-type*
 42 *pointer-type*
 43 *pointer-type:*
 44 *unmanaged-type* *
 45 void *
 46 *unmanaged-type:*
 47 *type*

C# LANGUAGE SPECIFICATION

1 *primary-no-array-creation-expression:*
2 ...
3 *pointer-member-access*
4 *pointer-element-access*
5 *sizeof-expression*
6 *unary-expression:*
7 ...
8 *pointer-indirection-expression*
9 *addressof-expression*
10 *pointer-indirection-expression:*
11 * *unary-expression*
12 *pointer-member-access:*
13 *primary-expression* -> *identifier*
14 *pointer-element-access:*
15 *primary-no-array-creation-expression* [*expression*]
16 *addressof-expression:*
17 & *unary-expression*
18 *sizeof-expression:*
19 sizeof (*unmanaged-type*)
20 *embedded-statement:*
21 ...
22 *fixed-statement*
23 *fixed-statement:*
24 fixed (*pointer-type* *fixed-pointer-declarators*) *embedded-statement*
25 *fixed-pointer-declarators:*
26 *fixed-pointer-declarator*
27 *fixed-pointer-declarators* , *fixed-pointer-declarator*
28 *fixed-pointer-declarator:*
29 *identifier* = *fixed-pointer-initializer*
30 *fixed-pointer-initializer:*
31 & *variable-reference*
32 *expression*
33 *variable-initializer:*
34 *expression*
35 *array-initializer*
36 *stackalloc-initializer*
37 *stackalloc-initializer:*
38 stackalloc *unmanaged-type* [*expression*]
39 **End of informative text.**

B. Portability issues

This clause is informative.

This annex collects some information about portability that appears in this ECMA Standard.

B.1 Undefined behavior

A program that does not contain any occurrences of the `unsafe` modifier cannot exhibit any undefined behavior.

The behavior is undefined in the following circumstances:

1. When dereferencing the result of converting one pointer type to another, and the resulting pointer is not correctly aligned for the pointed-to type. (§25.4)
2. When the unary `*` operator is applied to a pointer containing an invalid value (§25.5.1).
3. When a pointer is subscripted to access an out-of-bounds element (§25.5.3).
4. Modifying objects of managed type through fixed pointers (§25.6)
5. The initial content of memory allocated by `stackalloc` (§25.7).

B.2 Implementation-defined behavior

A conforming implementation is required to document its choice of behavior in each of the areas listed in this clause. The following are implementation-defined:

1. The behavior when an identifier not in Normalization Form C is encountered (§9.4.2).
2. The values of any application parameters passed to `Main` by the host environment prior to application startup (§10.1).
3. The mechanism by which linkage to an external function is achieved (§17.5.7).
4. The impact of thread termination when no matching catch clause is found for an exception and the code that initially started that thread is reached. (§23.3)
5. The purpose of attribute target specifiers other than those identified by this standard (§24.2).
6. The mappings between pointers and integers (§25.4).
7. The effect of applying the unary `*` operator to a `null` pointer (§25.5.1).
8. The behavior when pointer arithmetic overflows the domain of the pointer type (§25.5.5).
9. The result of the `sizeof` operator for other than the pre-defined value types (§25.5.8).
10. The behavior of the `fixed` statement if the array expression is null or if the array has zero elements (§25.6).

- 1 11. The behavior of the `fixed` statement if the string expression is null (§25.6).
- 2 12. The value returned when a stack allocation of size zero is made (§25.7).

3 **B.3 Unspecified behavior**

- 4
- 5 1. The time at which the destructor (if any) for an object is run, once that object has become eligible for
- 6 destruction.
- 7 2. The value of the result when converting out-of-range values from `float` or `double` values to an integral
- 8 type in an unchecked context (§13.2.1).
- 9 3. The layout of arrays, except in an unsafe context (§14.5.10.2).
- 10 4. The exact timing of static field initialization (§17.4.5.1).
- 11 5. The order in which members are packed into a struct (§25.5.8).

12 **B.4 Other Issues**

- 13
- 14 1. The exact results of floating-point expression evaluation may vary from one implementation to another,
- 15 because an implementation is permitted to evaluate such expressions using a greater range and/or
- 16 precision than is required. (§11.1.5)
- 17 2. The CLI reserves certain signatures for compatibility with other programming languages. (§17.2.7)

18 **End of informative text.**

C. Naming guidelines

1

2 **This annex is informative.**

3 One of the most important elements of predictability and discoverability is the use of a consistent naming
4 pattern. Many of the common user questions don't even arise once these conventions are understood and
5 widely used. There are three elements to the naming guidelines:

- 6 1. Casing – use of the correct capitalization style
- 7 2. Mechanical – use nouns for classes, verbs for methods, etc.
- 8 3. Word choice – use consistent terms across class libraries.

9 The following section lays out rules for the first two elements, and some philosophy for the third.

10 **C.1 Capitalization styles**

11 The following section describes different ways of capitalizing identifiers.

12 **C.1.1 Pascal casing**

13 This convention capitalizes the first character of each word. For example:

14 `color BitConverter`

15 **C.1.2 Camel casing**

16 This convention capitalizes the first character of each word except the first word. For example:

17 `backgroundColor totalValueCount`

18 **C.1.3 All uppercase**

19 Only use all uppercase letters for an identifier if it contains an abbreviation. For example:

20 `System.IO`
21 `System.Windows.UI`

22 **C.1.4 Capitalization summary**

23 The following table summarizes the capitalization style for the different kinds of identifiers:

24

Type	Case	Notes
Class	PascalCase	
Attribute Class	PascalCase	Has a suffix of <code>Attribute</code>
Exception Class	PascalCase	Has a suffix of <code>Exception</code>
Constant	PascalCase	
Enum type	PascalCase	
Enum values	PascalCase	
Event	PascalCase	
Interface	PascalCase	Has a prefix of <code>I</code>
Local variable	camelCase	
Method	PascalCase	
Namespace	PascalCase	
Property	PascalCase	
Public Instance Field	PascalCase	Rarely used (use a property instead)
Protected Instance Field	camelCase	Rarely used (use a property instead)
Parameter	camelCase	

1

2 C.2 Word choice

- 3 • Do avoid using class names duplicated in heavily used namespaces. For example, don't use the
4 following for a class name.

5 `System` `Collections` `Forms` `UI`

- 6 • Do not use abbreviations in identifiers.
- 7 • If you must use abbreviations, do use camelCase for any abbreviation containing more than two
8 characters, even if this is not the usual abbreviation.

9 C.3 Namespaces

10 The general rule for namespace naming is: `CompanyName.TechnologyName`.

- 11 • Do avoid the possibility of two published namespaces having the same name, by prefixing namespace
12 names with a company name or other well-established brand. For example, `Microsoft.Office` for the
13 Office Automation classes provided by Microsoft.
- 14 • Do use PascalCase, and separate logical components with periods (as in
15 `Microsoft.Office.PowerPoint`). If your brand employs non-traditional casing, do follow the
16 casing defined by your brand, even if it deviates from normal namespace casing (for example,
17 `NEXT.webObjects`, and `ee.cummings`).
- 18 • Do use plural namespace names where appropriate. For example, use `System.Collections` rather
19 than `System.Collection`. Exceptions to this rule are brand names and abbreviations. For example,
20 use `System.IO` not `System.IOs`.
- 21 • Do not have namespaces and classes with the same name.

22 C.4 Classes

- 23 • Do name classes with nouns or noun phrases.

- 1 • Do use PascalCase.
- 2 • Do use sparingly, abbreviations in class names.
- 3 • Do not use any prefix (such as “C”, for example). Where possible, avoid starting with the letter “I”,
- 4 since that is the recommended prefix for interface names. If you must start with that letter, make sure the
- 5 second character is lowercase, as in `IdentityStore`.

- 6 • Do not use any underscores.

```
7     public class FileStream { ... }
8     public class Button { ... }
9     public class String { ... }
```

10 C.5 Interfaces

- 11 • Do name interfaces with nouns or noun phrases, or adjectives describing behavior. For example,
- 12 `IComponent` (descriptive noun), `ICustomAttributeProvider` (noun phrase), and
- 13 `IPersistable` (adjective).
- 14 • Do use PascalCase.
- 15 • Do use sparingly, abbreviations in interface names.
- 16 • Do not use any underscores.
- 17 • Do prefix interface names with the letter “I”, to indicate that the type is an interface.
- 18 • Do use similar names when defining a class/interface pair where the class is a standard implementation
- 19 of the interface. The names should differ only by the “I” prefix in the interface name. This approach is
- 20 used for the interface `IComponent` and its standard implementation, `Component`.

```
21     public interface IComponent { ... }
22     public class Component : IComponent { ... }
23     public interface IServiceProvider { ... }
24     public interface IFormatable { ... }
```

25 C.6 Enums

- 26 • Do use PascalCase for enums.
- 27 • Do use PascalCase for enum value names.
- 28 • Do use sparingly, abbreviations in enum names.
- 29 • Do not use a family-name prefix on enum.
- 30 • Do not use any “Enum” suffix on enum types.
- 31 • Do use a singular name for enums
- 32 • Do use a plural name for bit fields
- 33 • Do define enumerated values using an enum if they are used in a parameter or property. This gives
- 34 development tools a chance at knowing the possible values for a property or parameter.

```
35     public enum FileMode{
36         Create,
37         CreateNew,
38         Open,
39         OpenOrCreate,
40         Truncate
41     }
```

- 42 • Do use the `Flags` custom attribute if the numeric values are meant to be bitwise ORed together

```
1     [Flags]
2     public enum Bindings {
3         CreateInstance,
4         DefaultBinding,
5         ExcatBinding,
6         GetField,
7         GetProperty,
8         IgnoreCase,
9         InvokeMethod,
10        NonPublic,
11        OABinding,
12        SetField
13        SetProperty,
14        Static
15    }
```

- 16 • Do use `int` as the underlying type of an enum. (An exception to this rule is if the enum represents flags and there are more than 32 flags, or the enum may grow to that many flags in the future, or the type needs to be different from `int` for backward compatibility.)
- 19 • Do use enums only if the value can be completely expressed as a set of bit flags. Do not use enums for open sets (such as operating system version).

21 C.7 Static fields

- 22 • Do name static members with nouns, noun phrases, or abbreviations for nouns.
- 23 • Do name static members using PascalCase.
- 24 • Do not use Hungarian-type prefixes on static member names.

25 C.8 Parameters

- 26 • Do use descriptive names such that a parameter's name and type clearly imply its meaning.
- 27 • Do name parameters using camelCase.
- 28 • Do prefer names based on a parameter's meaning, to names based on the parameter's type. It is likely that development tools will provide the information about type in a convenient way, so the parameter name can be put to better use describing semantics rather than type.
- 31 • Do not reserve parameters for future use. If more data is need in the next version, a new overload can be added.
- 33 • Do not use Hungarian-type prefixes.

```
34     Type GetType (string typeName)
35     string Format (string format, object [] args)
```

36 C.9 Methods

- 37 • Do name methods with verbs or verb phrases.
- 38 • Do name methods with PascalCase
- 39 RemoveAll(), GetCharArray(), Invoke()

40 C.10 Properties

- 41 • Do name properties using noun or noun phrases
- 42 • Do name properties with PascalCase
- 43 • Consider having a property with the same as a type. When declaring a property with the same name as a type, also make the type of the property be that type. In other words, the following is okay
- 44

```

1     public enum Color {...}
2     public class Control {
3         public Color Color { get {...} set {...} }
4     }

```

5 but this is not

```

6     public enum Color {...}
7     public class Control {
8         public int Color { get {...} set {...} }
9     }

```

10 In the latter case, it will not be possible to refer to the members of the Color enum because `Color.Xxx`
 11 will be interpreted as being a member access that first gets the value of the `Color` property (of type
 12 `int`) and then accesses a member of that value (which would have to be an instance member of
 13 `System.Int32`).

14 C.11 Events

- 15 • Do name event handlers with the “EventHandler” suffix.

```
16     public delegate void MouseEventHandler(object sender, MouseEventArgs e);
```

- 17 • Do use two parameters named `sender` and `e`. The sender parameter represents the object that raised the
 18 event, and this parameter is always of type `object`, even if it is possible to employ a more specific type.
 19 The state associated with the event is encapsulated in an instance `e` of an event class. Use an appropriate
 20 and specific event class for its type.

```
21     public delegate void MouseEventHandler(object sender, MouseEventArgs e);
```

- 22 • Do name event argument classes with the “EventArgs” suffix.

```

23     public class MouseEventArgs : EventArgs {
24         int x;
25         int y;
26         public MouseEventArgs(int x, int y)
27             { this.x = x; this.y = y; }
28         public int X { get { return x; } }
29         public int Y { get { return y; } }
30     }

```

- 31 • Do name event names that have a concept of pre- and post-operation using the present and past tense (do
 32 not use `BeforeXxx/AfterXxx` pattern). For example, a close event that could be canceled would have a
 33 `Closing` and `Closed` event.

```

34     public event ControlEventHandler ControlAdded {
35         //..
36     }

```

- 37 • Consider naming events with a verb.

38 C.12 Case sensitivity

- 39 • Don't use names that require case sensitivity. Components might need to be usable from both case-
 40 sensitive and case-insensitive languages. Since case-insensitive languages cannot distinguish between
 41 two names within the same context that differ only by case, components must avoid this situation.

42 Examples of what not to do:

- 43 • Don't have two namespaces whose names differ only by case.

```

44     namespace ee.cummings;
45     namespace Ee.Cummings;

```

- 46 • Don't have a method with two parameters whose names differ only by case.

```
47     void F(string a, string A)
```

- 1 • Don't have a namespace with two types whose names differ only by case.

```
2     System.Windows.Point p;  
3     System.Windows.POINT pp;
```

- 4 • Don't have a type with two properties whose names differ only by case.

```
5     int F {get, set};  
6     int F {get, set}
```

- 7 • Don't have a type with two methods whose names differ only by case.

```
8     void f();  
9     void F();
```

10 C.13 Avoiding type name confusion

11 Different languages use different names to identify the fundamental managed types, so in a multi-language
12 environment, designers must take care to avoid language-specific terminology. This section describes a set
13 of rules that help avoid type name confusion.

- 14 • Do use semantically interesting names rather than type names.
- 15 • In the rare case that a parameter has no semantic meaning beyond its type, use a generic name. For
16 example, a class that supports writing a variety of data types into a stream might have:

```
17     void write(double value);  
18     void write(float value);  
19     void write(long value);  
20     void write(int value);  
21     void write(short value);
```

22 rather than a language-specific alternative such as:

```
23     void write(double doublevalue);  
24     void write(float floatValue);  
25     void write(long longvalue);  
26     void write(int intValue);  
27     void write(short shortvalue);
```

- 28 • In the extremely rare case that it is necessary to have a uniquely named method for each fundamental
29 data type, do use the following universal type names: `Sbyte`, `Byte`, `Int16`, `UInt16`, `Int32`, `UInt32`,
30 `Int64`, `UInt64`, `Single`, `Double`, `Boolean`, `Char`, `String`, and `Object`. For example, a class that
31 supports reading a variety of data types from a stream might have:

```
32     double ReadDouble();  
33     float ReadSingle();  
34     long ReadInt64();  
35     int ReadInt32();  
36     short ReadInt16();
```

37 rather than a language-specific alternative such as:

```
38     double ReadDouble();  
39     float ReadFloat();  
40     long ReadLong();  
41     int ReadInt();  
42     short ReadShort();
```

43 **End of informative text.**

D. Standard Library

1

2 A conforming C# implementation shall provide a set of types having specific semantics. For convenience,
 3 these types and their members are listed here, in alphabetical order. For a formal definition of these types
 4 and their members, refer to ECMA-xxx, 1st Edition, December 2001, *Common Language Infrastructure*
 5 *(CLI), Partition IV; Base Class Library (BCL), Extended Numerics Library, and Extended Array Library,*
 6 which are included by reference in this ECMA Standard.

7 **This rest of this clause is informative.**

8

9 // Namespace: System, Library: BCL

10 public class ApplicationException: Exception

11 {

12 public ApplicationException();

13 public ApplicationException(string message);

14 public ApplicationException(string message, Exception innerException);

15 }

16

17 // Namespace: System, Library: BCL

18 public class ArgumentException: SystemException

19 {

20 public ArgumentException();

21 public ArgumentException(string message);

22 public ArgumentException(string message, Exception innerException);

23 public ArgumentException(string message, string paramName, Exception
 24 innerException);

25 public ArgumentException(string message, string paramName);

26 public virtual string ParamName { get; }

27 }

28

29 // Namespace: System, Library: BCL

30 public class ArgumentNullException: ArgumentException

31 {

32 public ArgumentNullException();

33 public ArgumentNullException(string paramName);

34 public ArgumentNullException(string paramName, string message);

35 }

36

37 // Namespace: System, Library: BCL

38 public class ArgumentOutOfRangeException: ArgumentException

39 {

40 public ArgumentOutOfRangeException();

C# LANGUAGE SPECIFICATION

```
1     public ArgumentOutOfRangeException(string paramName);
2     public ArgumentOutOfRangeException(string paramName, string message);
3     public ArgumentOutOfRangeException(string paramName, object actualValue,
4         string message);
5     public virtual object ActualValue { get; }
6 }
7
8 // Namespace: System, Library: BCL
9 public class ArithmeticException: SystemException
10 {
11     public ArithmeticException();
12     public ArithmeticException(string message);
13     public ArithmeticException(string message, Exception innerException);
14 }
15
16 // Namespace: System, Library: BCL
17 public abstract class Array: ICloneable, ICollection, IEnumerable, IList
18 {
19     protected Array();
20     public static int BinarySearch(Array array, object value);
21     public static int BinarySearch(Array array, int index, int length, object
22         value);
23     public static int BinarySearch(Array array, object value, IComparer
24         comparer);
25     public static int BinarySearch(Array array, int index, int length, object
26         value, IComparer comparer);
27     public static void Clear(Array array, int index, int length);
28     public virtual object Clone();
29     public static void Copy(Array sourceArray, Array destinationArray, int
30         length);
31     public static void Copy(Array sourceArray, int sourceIndex, Array
32         destinationArray, int destinationIndex, int length);
33     public virtual void CopyTo(Array array, int index);
34     public static Array CreateInstance(Type elementType, int length);
35     public static Array CreateInstance(Type elementType, int length1, int
36         length2);
37     public static Array CreateInstance(Type elementType, int length1, int
38         length2, int length3);
39     public static Array CreateInstance(Type elementType, int[] lengths);
40     public static Array CreateInstance(Type elementType, int[] lengths, int[]
41         lowerBounds);
42     public virtual IEnumerator GetEnumerator();
43     public object GetValue(int[] indices);
44     public object GetValue(int index);
```



```

1 public object GetValue(int index1, int index2);
2 public object GetValue(int index1, int index2, int index3);
3 public static int IndexOf(Array array, object value);
4 public static int IndexOf(Array array, object value, int startIndex);
5 public static int IndexOf(Array array, object value, int startIndex, int
6     count);
7 public static int LastIndexOf(Array array, object value);
8 public static int LastIndexOf(Array array, object value, int startIndex);
9 public static int LastIndexOf(Array array, object value, int startIndex,
10     int count);
11 public static void Reverse(Array array);
12 public static void Reverse(Array array, int index, int length);
13 public void SetValue(object value, int index);
14 public void SetValue(object value, int index1, int index2);
15 public void SetValue(object value, int index1, int index2, int index3);
16 public void SetValue(object value, int[] indices);
17 public static void Sort(Array array);
18 public static void Sort(Array keys, Array items);
19 public static void Sort(Array array, int index, int length);
20 public static void Sort(Array keys, Array items, int index, int length);
21 public static void Sort(Array array, IComparer comparer);
22 public static void Sort(Array keys, Array items, IComparer comparer);
23 public static void Sort(Array array, int index, int length, IComparer
24     comparer);
25 public static void Sort(Array keys, Array items, int index, int length,
26     IComparer comparer);
27 int IList.Add(object value);
28 void IList.Clear();
29 bool IList.Contains(object value);
30 int IList.IndexOf(object value);
31 void IList.Insert(int index, object value);
32 void IList.Remove(object value);
33 void IList.RemoveAt(int index);
34 bool IList.IsFixedSize { get; }
35 bool IList.IsReadOnly { get; }
36 bool ICollection.IsSynchronized { get; }
37 public int Length { get; }
38 public long LongLength {get;}
39 public int Rank { get; }
40 object ICollection.SyncRoot { get; }
41 int ICollection.Count { get; }
42 public virtual object this[int index] { get; set; }
43 }
44

```

C# LANGUAGE SPECIFICATION

```
1 // Namespace: System.Collections, Library: BCL
2 public class ArrayList: ICloneable, ICollection, IEnumerable, IList
3 {
4     public ArrayList();
5     public ArrayList(int capacity);
6     public ArrayList(ICollection c);
7     public static ArrayList Adapter(IList list);
8     public virtual int Add(object value);
9     public virtual void AddRange(ICollection c);
10    public virtual int BinarySearch(object value, IComparer comparer);
11    public virtual int BinarySearch(object value);
12    public virtual int BinarySearch(int index, int count, object value,
13        IComparer comparer);
14    public virtual void Clear();
15    public virtual object Clone();
16    public virtual bool Contains(object item);
17    public virtual void CopyTo(Array array, int arrayIndex);
18    public virtual void CopyTo(int index, Array array, int arrayIndex, int
19        count);
20    public virtual void CopyTo(Array array);
21    public static ArrayList FixedSize(ArrayList list);
22    public virtual IEnumerator GetEnumerator();
23    public virtual IEnumerator GetEnumerator(int index, int count);
24    public virtual ArrayList GetRange(int index, int count);
25    public virtual int IndexOf(object value);
26    public virtual int IndexOf(object value, int startIndex, int count);
27    public virtual int IndexOf(object value, int startIndex);
28    public virtual void Insert(int index, object value);
29    public virtual void InsertRange(int index, ICollection c);
30    public virtual int LastIndexOf(object value, int startIndex, int count);
31    public virtual int LastIndexOf(object value, int startIndex);
32    public virtual int LastIndexOf(object value);
33    public static ArrayList ReadOnly(ArrayList list);
34    public virtual void Remove(object obj);
35    public virtual void RemoveAt(int index);
36    public virtual void RemoveRange(int index, int count);
37    public static ArrayList Repeat(object value, int count);
38    public virtual void Reverse(int index, int count);
39    public virtual void Reverse();
40    public virtual void SetRange(int index, ICollection c);
41    public virtual void Sort(int index, int count, IComparer comparer);
42    public virtual void Sort(IComparer comparer);
43    public virtual void Sort();
44    public static ArrayList Synchronized(ArrayList list);
```

```

1     public virtual Array ToArray(Type type);
2     public virtual object[] ToArray();
3     public virtual void TrimToSize();
4     public virtual int Capacity { get; set; }
5     int ICollection.Count { get; }
6     public virtual int Count { get; }
7     bool IList.IsFixedSize { get; }
8     public virtual bool IsFixedSize { get; }
9     bool IList.IsReadOnly { get; }
10    public virtual bool IsReadOnly { get; }
11    bool ICollection.IsSynchronized { get; }
12    public virtual bool IsSynchronized { get; }
13    public virtual object this[int index] { get; set; }
14    object ICollection.SyncRoot { get; }
15    public virtual object SyncRoot { get; }
16 }
17
18 // Namespace: System, Library: BCL
19 public class ArrayTypeMismatchException: SystemException
20 {
21     public ArrayTypeMismatchException();
22     public ArrayTypeMismatchException(string message);
23     public ArrayTypeMismatchException(string message, Exception innerException);
24 }
25
26 // Namespace: System.Text, Library: BCL
27 public class ASCIIEncoding: Encoding
28 {
29     public ASCIIEncoding();
30     public override int GetByteCount(char[] chars, int index, int count);
31     public override int GetByteCount(string chars);
32     public override int GetBytes(string chars, int charIndex, int charCount,
33         byte[] bytes, int byteIndex);
34     public override int GetBytes(char[] chars, int charIndex, int charCount,
35         byte[] bytes, int byteIndex);
36     public override int GetCharCount(byte[] bytes, int index, int count);
37     public override int GetChars(byte[] bytes, int byteIndex, int byteCount,
38         char[] chars, int charIndex);
39     public override int GetMaxByteCount(int charCount);
40     public override int GetMaxCharCount(int byteCount);
41     public override string GetString(byte[] bytes, int byteIndex, int
42         byteCount);
43     public override string GetString(byte[] bytes);
44 }

```

C# LANGUAGE SPECIFICATION

```
1
2 // Namespace: System, Library: BCL
3 public delegate void AsyncCallback(IAsyncResult ar);
4
5 // Namespace: System, Library: BCL
6 public abstract class Attribute
7 {
8     protected Attribute();
9     public override bool Equals(object obj);
10    public override int GetHashCode();
11 }
12
13 // Namespace: System, Library: BCL
14 public enum AttributeTargets
15 {
16     All = Assembly | 0x2 | Class | Struct | Enum | Constructor | Method |
17         Property | Field | Event | Interface | Parameter | Delegate |
18         ReturnValue,
19     Assembly = 0x1,
20     Class = 0x4,
21     Constructor = 0x20,
22     Delegate = 0x1000,
23     Enum = 0x10,
24     Event = 0x200,
25     Field = 0x100,
26     Interface = 0x400,
27     Method = 0x40,
28     Parameter = 0x800,
29     Property = 0x80,
30     ReturnValue = 0x2000,
31     Struct = 0x8,
32 }
33
34 // Namespace: System, Library: BCL
35 public sealed class AttributeUsageAttribute: Attribute
36 {
37     public AttributeUsageAttribute(AttributeTargets validOn);
38     public bool AllowMultiple { get; set; }
39     public bool Inherited { get; set; }
40     public AttributeTargets ValidOn { get; }
41 }
42
43 // Namespace: System, Library: BCL
44 public struct Boolean: IComparable
```

```

1  {
2      public static readonly string FalseString;
3      public static readonly string TrueString;
4      public int CompareTo(object obj);
5      public override bool Equals(object obj);
6      public override int GetHashCode();
7      public static bool Parse(string value);
8      public string ToString(IFormatProvider provider);
9      public override string ToString();
10 }
11
12 // Namespace: System, Library: BCL
13 public struct Byte: IComparable, IFormattable
14 {
15     public const byte MaxValue = 255;
16     public const byte MinValue = 0;
17     public int CompareTo(object value);
18     public override bool Equals(object obj);
19     public override int GetHashCode();
20     public static byte Parse(string s);
21     public static byte Parse(string s, NumberStyles style);
22     public static byte Parse(string s, IFormatProvider provider);
23     public static byte Parse(string s, NumberStyles style, IFormatProvider
24         provider);
25     public string ToString(IFormatProvider provider);
26     public string ToString(string format, IFormatProvider provider);
27     public override string ToString();
28     public string ToString(string format);
29 }
30
31 // Namespace: System, Library: BCL
32 public struct Char: IComparable
33 {
34     public const char MaxValue = (char)0xFFFF;
35     public const char MinValue = (char)0x0;
36     public int CompareTo(object value);
37     public override bool Equals(object obj);
38     public override int GetHashCode();
39     public static double GetNumericValue(char c);
40     public static double GetNumericValue(string s, int index);
41     public static UnicodeCategory GetUnicodeCategory(char c);
42     public static UnicodeCategory GetUnicodeCategory(string s, int index);
43     public static bool IsControl(char c);
44     public static bool IsControl(string s, int index);

```

C# LANGUAGE SPECIFICATION

```
1     public static bool IsDigit(char c);
2     public static bool IsDigit(string s, int index);
3     public static bool IsLetter(char c);
4     public static bool IsLetter(string s, int index);
5     public static bool IsLetterOrDigit(char c);
6     public static bool IsLetterOrDigit(string s, int index);
7     public static bool IsLower(char c);
8     public static bool IsLower(string s, int index);
9     public static bool IsNumber(char c);
10    public static bool IsNumber(string s, int index);
11    public static bool IsPunctuation(char c);
12    public static bool IsPunctuation(string s, int index);
13    public static bool IsSeparator(char c);
14    public static bool IsSeparator(string s, int index);
15    public static bool IsSurrogate(char c);
16    public static bool IsSurrogate(string s, int index);
17    public static bool Issymbol(char c);
18    public static bool Issymbol(string s, int index);
19    public static bool IsUpper(char c);
20    public static bool IsUpper(string s, int index);
21    public static bool IsWhiteSpace(char c);
22    public static bool IsWhiteSpace(string s, int index);
23    public static char Parse(string s);
24    public static char ToLower(char c);
25    public string ToString(IFormatProvider provider);
26    public override string ToString();
27    public static char ToUpper(char c);
28 }
29
30 // Namespace: System, Library: BCL
31 public sealed class CharEnumerator: ICloneable, IEnumerator
32 {
33     public object Clone();
34     public bool MoveNext();
35     public void Reset();
36     public char Current { get; }
37     object IEnumerator.Current { get; }
38 }
39
40 // Namespace: System, Library: BCL
41 public sealed class CLSCompliantAttribute: Attribute
42 {
43     public CLSCompliantAttribute(bool isCompliant);
44     public bool IsCompliant { get; }
```

```
1  }
2
3  // Namespace: System.Security, Library: BCL
4  public abstract class CodeAccessPermission: IPermission
5  {
6      protected CodeAccessPermission();
7      public void Assert();
8      public abstract IPermission Copy();
9      public void Demand();
10     public void Deny();
11     public abstract void FromXml(SecurityElement elem);
12     public abstract IPermission Intersect(IPermission target);
13     public abstract bool IsSubsetOf(IPermission target);
14     public override string ToString();
15     public abstract SecurityElement ToXml();
16     public virtual IPermission Union(IPermission other);
17 }
18
19 // Namespace: System.Security.Permissions, Library: BCL
20 public abstract class CodeAccessSecurityAttribute: SecurityAttribute
21 {
22     protected CodeAccessSecurityAttribute();
23     public CodeAccessSecurityAttribute(SecurityAction action);
24 }
25
26 // Namespace: System.Collections, Library: BCL
27 public sealed class Comparer: IComparer
28 {
29     public static readonly Comparer Default;
30     public int Compare(object a, object b);
31 }
32
33 // Namespace: System.Diagnostics, Library: BCL
34 public sealed class ConditionalAttribute: Attribute
35 {
36     public ConditionalAttribute(string conditionString);
37     public string ConditionString { get; }
38 }
39
40 // Namespace: System, Library: BCL
41 public sealed class Console
42 {
43     public static Stream OpenStandardError();
44     public static Stream OpenStandardError(int bufferSize);
```

C# LANGUAGE SPECIFICATION

```
1     public static Stream OpenStandardInput();
2     public static Stream OpenStandardInput(int bufferSize);
3     public static Stream OpenStandardOutput();
4     public static Stream OpenStandardOutput(int bufferSize);
5     public static int Read();
6     public static string ReadLine();
7     public static void SetError(TextWriter newError);
8     public static void SetIn(TextReader newIn);
9     public static void SetOut(TextWriter newOut);
10    public static void write(string format, object arg0);
11    public static void write(string format, object arg0, object arg1);
12    public static void write(string format, object arg0, object arg1, object
13        arg2);
14    public static void write(string format, params object[] arg);
15    public static void write(bool value);
16    public static void write(char value);
17    public static void write(char[] buffer);
18    public static void write(char[] buffer, int index, int count);
19    public static void write(double value);
20    public static void write(decimal value);
21    public static void write(float value);
22    public static void write(int value);
23    public static void write(uint value);
24    public static void write(long value);
25    public static void write(ulong value);
26    public static void write(object value);
27    public static void write(string value);
28    public static void WriteLine();
29    public static void WriteLine(bool value);
30    public static void WriteLine(char value);
31    public static void WriteLine(char[] buffer);
32    public static void WriteLine(char[] buffer, int index, int count);
33    public static void WriteLine(decimal value);
34    public static void WriteLine(double value);
35    public static void WriteLine(float value);
36    public static void WriteLine(int value);
37    public static void WriteLine(uint value);
38    public static void WriteLine(long value);
39    public static void WriteLine(ulong value);
40    public static void WriteLine(object value);
41    public static void WriteLine(string value);
42    public static void WriteLine(string format, object arg0);
43    public static void WriteLine(string format, object arg0, object arg1);
44    public static void WriteLine(string format, object arg0, object arg1,
```



```
1     object arg2);
2     public static void WriteLine(string format, params object[] arg);
3     public static TextWriter Error { get; }
4     public static TextReader In { get; }
5     public static TextWriter Out { get; }
6 }
7
8 // Namespace: System, Library: BCL
9 public sealed class Convert
10 {
11     public static bool ToBoolean(bool value);
12     public static bool ToBoolean(sbyte value);
13     public static bool ToBoolean(byte value);
14     public static bool ToBoolean(short value);
15     public static bool ToBoolean(ushort value);
16     public static bool ToBoolean(int value);
17     public static bool ToBoolean(uint value);
18     public static bool ToBoolean(long value);
19     public static bool ToBoolean(ulong value);
20     public static bool ToBoolean(string value);
21     public static bool ToBoolean(float value);
22     public static bool ToBoolean(double value);
23     public static bool ToBoolean(decimal value);
24     public static byte ToByte(bool value);
25     public static byte ToByte(byte value);
26     public static byte ToByte(char value);
27     public static byte ToByte(sbyte value);
28     public static byte ToByte(short value);
29     public static byte ToByte(ushort value);
30     public static byte ToByte(int value);
31     public static byte ToByte(uint value);
32     public static byte ToByte(long value);
33     public static byte ToByte(ulong value);
34     public static byte ToByte(float value);
35     public static byte ToByte(double value);
36     public static byte ToByte(decimal value);
37     public static byte ToByte(string value);
38     public static byte ToByte(string value, IFormatProvider provider);
39     public static char ToChar(char value);
40     public static char ToChar(sbyte value);
41     public static char ToChar(byte value);
42     public static char ToChar(short value);
43     public static char ToChar(ushort value);
44     public static char ToChar(int value);
```

C# LANGUAGE SPECIFICATION

```
1     public static char ToChar(uint value);
2     public static char ToChar(long value);
3     public static char ToChar(ulong value);
4     public static char ToChar(string value);
5     public static DateTime ToDateTime(DateTime value);
6     public static DateTime ToDateTime(string value);
7     public static DateTime ToDateTime(string value, IFormatProvider provider);
8     public static decimal ToDecimal(sbyte value);
9     public static decimal ToDecimal(byte value);
10    public static decimal ToDecimal(short value);
11    public static decimal ToDecimal(ushort value);
12    public static decimal ToDecimal(int value);
13    public static decimal ToDecimal(uint value);
14    public static decimal ToDecimal(long value);
15    public static decimal ToDecimal(ulong value);
16    public static decimal ToDecimal(float value);
17    public static decimal ToDecimal(double value);
18    public static decimal ToDecimal(string value);
19    public static decimal ToDecimal(string value, IFormatProvider provider);
20    public static decimal ToDecimal(decimal value);
21    public static decimal ToDecimal(bool value);
22    public static double ToDouble(sbyte value);
23    public static double ToDouble(byte value);
24    public static double ToDouble(short value);
25    public static double ToDouble(ushort value);
26    public static double ToDouble(int value);
27    public static double ToDouble(uint value);
28    public static double ToDouble(long value);
29    public static double ToDouble(ulong value);
30    public static double ToDouble(float value);
31    public static double ToDouble(double value);
32    public static double ToDouble(decimal value);
33    public static double ToDouble(string value);
34    public static double ToDouble(string value, IFormatProvider provider);
35    public static double ToDouble(bool value);
36    public static short.ToInt16(bool value);
37    public static short.ToInt16(char value);
38    public static short.ToInt16(sbyte value);
39    public static short.ToInt16(byte value);
40    public static short.ToInt16(ushort value);
41    public static short.ToInt16(int value);
42    public static short.ToInt16(uint value);
43    public static short.ToInt16(short value);
44    public static short.ToInt16(long value);
```

```
1 public static short ToInt16(ulong value);
2 public static short ToInt16(float value);
3 public static short ToInt16(double value);
4 public static short ToInt16(decimal value);
5 public static short ToInt16(string value);
6 public static short ToInt16(string value, IFormatProvider provider);
7 public static int ToInt32(bool value);
8 public static int ToInt32(char value);
9 public static int ToInt32(sbyte value);
10 public static int ToInt32(byte value);
11 public static int ToInt32(short value);
12 public static int ToInt32(ushort value);
13 public static int ToInt32(uint value);
14 public static int ToInt32(int value);
15 public static int ToInt32(long value);
16 public static int ToInt32(ulong value);
17 public static int ToInt32(float value);
18 public static int ToInt32(double value);
19 public static int ToInt32(decimal value);
20 public static int ToInt32(string value);
21 public static int ToInt32(string value, IFormatProvider provider);
22 public static long ToInt64(bool value);
23 public static long ToInt64(char value);
24 public static long ToInt64(sbyte value);
25 public static long ToInt64(byte value);
26 public static long ToInt64(short value);
27 public static long ToInt64(ushort value);
28 public static long ToInt64(int value);
29 public static long ToInt64(uint value);
30 public static long ToInt64(ulong value);
31 public static long ToInt64(long value);
32 public static long ToInt64(float value);
33 public static long ToInt64(double value);
34 public static long ToInt64(decimal value);
35 public static long ToInt64(string value);
36 public static long ToInt64(string value, IFormatProvider provider);
37 public static sbyte ToSByte(bool value);
38 public static sbyte ToSByte(sbyte value);
39 public static sbyte ToSByte(char value);
40 public static sbyte ToSByte(byte value);
41 public static sbyte ToSByte(short value);
42 public static sbyte ToSByte(ushort value);
43 public static sbyte ToSByte(int value);
44 public static sbyte ToSByte(uint value);
```

C# LANGUAGE SPECIFICATION

```
1     public static sbyte ToSByte(long value);
2     public static sbyte ToSByte(ulong value);
3     public static sbyte ToSByte(float value);
4     public static sbyte ToSByte(double value);
5     public static sbyte ToSByte(decimal value);
6     public static sbyte ToSByte(string value);
7     public static sbyte ToSByte(string value, IFormatProvider provider);
8     public static float ToSingle(sbyte value);
9     public static float ToSingle(byte value);
10    public static float ToSingle(short value);
11    public static float ToSingle(ushort value);
12    public static float ToSingle(int value);
13    public static float ToSingle(uint value);
14    public static float ToSingle(long value);
15    public static float ToSingle(ulong value);
16    public static float ToSingle(float value);
17    public static float ToSingle(double value);
18    public static float ToSingle(decimal value);
19    public static float ToSingle(string value);
20    public static float ToSingle(string value, IFormatProvider provider);
21    public static float ToSingle(bool value);
22    public static string ToString(bool value);
23    public static string ToString(char value);
24    public static string ToString(sbyte value);
25    public static string ToString(sbyte value, IFormatProvider provider);
26    public static string ToString(byte value);
27    public static string ToString(byte value, IFormatProvider provider);
28    public static string ToString(short value);
29    public static string ToString(short value, IFormatProvider provider);
30    public static string ToString(ushort value);
31    public static string ToString(ushort value, IFormatProvider provider);
32    public static string ToString(int value);
33    public static string ToString(int value, IFormatProvider provider);
34    public static string ToString(uint value);
35    public static string ToString(uint value, IFormatProvider provider);
36    public static string ToString(long value);
37    public static string ToString(long value, IFormatProvider provider);
38    public static string ToString(ulong value);
39    public static string ToString(ulong value, IFormatProvider provider);
40    public static string ToString(float value);
41    public static string ToString(float value, IFormatProvider provider);
42    public static string ToString(double value);
43    public static string ToString(double value, IFormatProvider provider);
44    public static string ToString(decimal value);
```

```
1 public static string ToString(decimal value, IFormatProvider provider);
2 public static string ToString(DateTime value);
3 public static string ToString(DateTime value, IFormatProvider provider);
4 public static string ToString(string value);
5 public static ushort ToUInt16(bool value);
6 public static ushort ToUInt16(char value);
7 public static ushort ToUInt16(sbyte value);
8 public static ushort ToUInt16(byte value);
9 public static ushort ToUInt16(short value);
10 public static ushort ToUInt16(int value);
11 public static ushort ToUInt16(ushort value);
12 public static ushort ToUInt16(uint value);
13 public static ushort ToUInt16(long value);
14 public static ushort ToUInt16(ulong value);
15 public static ushort ToUInt16(float value);
16 public static ushort ToUInt16(double value);
17 public static ushort ToUInt16(decimal value);
18 public static ushort ToUInt16(string value);
19 public static ushort ToUInt16(string value, IFormatProvider provider);
20 public static uint ToUInt32(bool value);
21 public static uint ToUInt32(char value);
22 public static uint ToUInt32(sbyte value);
23 public static uint ToUInt32(byte value);
24 public static uint ToUInt32(short value);
25 public static uint ToUInt32(ushort value);
26 public static uint ToUInt32(int value);
27 public static uint ToUInt32(uint value);
28 public static uint ToUInt32(long value);
29 public static uint ToUInt32(ulong value);
30 public static uint ToUInt32(float value);
31 public static uint ToUInt32(double value);
32 public static uint ToUInt32(decimal value);
33 public static uint ToUInt32(string value);
34 public static uint ToUInt32(string value, IFormatProvider provider);
35 public static ulong ToUInt64(bool value);
36 public static ulong ToUInt64(char value);
37 public static ulong ToUInt64(sbyte value);
38 public static ulong ToUInt64(byte value);
39 public static ulong ToUInt64(short value);
40 public static ulong ToUInt64(ushort value);
41 public static ulong ToUInt64(int value);
42 public static ulong ToUInt64(uint value);
43 public static ulong ToUInt64(long value);
44 public static ulong ToUInt64(ulong value);
```

C# LANGUAGE SPECIFICATION

```
1     public static ulong ToUInt64(float value);
2     public static ulong ToUInt64(double value);
3     public static ulong ToUInt64(decimal value);
4     public static ulong ToUInt64(string value);
5     public static ulong ToUInt64(string value, IFormatProvider provider);
6 }
7
8 // Namespace: System, Library: BCL
9 public struct DateTime: IComparable, IFormattable
10 {
11     public DateTime(long ticks);
12     public DateTime(int year, int month, int day);
13     public DateTime(int year, int month, int day, int hour, int minute, int
14         second);
15     public DateTime(int year, int month, int day, int hour, int minute, int
16         second, int millisecond);
17     public static readonly DateTime MaxValue;
18     public static readonly DateTime MinValue;
19     public DateTime Add(TimeSpan value);
20     public DateTime AddDays(double value);
21     public DateTime AddHours(double value);
22     public DateTime AddMilliseconds(double value);
23     public DateTime AddMinutes(double value);
24     public DateTime AddMonths(int months);
25     public DateTime AddSeconds(double value);
26     public DateTime AddTicks(long value);
27     public DateTime AddYears(int value);
28     public static int Compare(DateTime t1, DateTime t2);
29     public int CompareTo(object value);
30     public static int DaysInMonth(int year, int month);
31     public override bool Equals(object value);
32     public static bool Equals(DateTime t1, DateTime t2);
33     public override int GetHashCode();
34     public static bool IsLeapYear(int year);
35     public static DateTime operator +(DateTime d, TimeSpan t);
36     public static bool operator ==(DateTime d1, DateTime d2);
37     public static bool operator >(DateTime t1, DateTime t2);
38     public static bool operator >=(DateTime t1, DateTime t2);
39     public static bool operator !=(DateTime d1, DateTime d2);
40     public static bool operator <(DateTime t1, DateTime t2);
41     public static bool operator <=(DateTime t1, DateTime t2);
42     public static DateTime operator -(DateTime d, TimeSpan t);
43     public static TimeSpan operator -(DateTime d1, DateTime d2);
44     public static DateTime Parse(string s);
```

```

1     public static DateTime Parse(string s, IFormatProvider provider);
2     public static DateTime Parse(string s, IFormatProvider provider,
3         DateTimeStyles styles);
4     public static DateTime ParseExact(string s, string format, IFormatProvider
5         provider);
6     public static DateTime ParseExact(string s, string format, IFormatProvider
7         provider, DateTimeStyles style);
8     public static DateTime ParseExact(string s, string[] formats,
9         IFormatProvider provider, DateTimeStyles style);
10    public TimeSpan Subtract(DateTime value);
11    public DateTime Subtract(TimeSpan value);
12    public DateTime ToLocalTime();
13    public string ToLongDateString();
14    public string ToLongTimeString();
15    public string ToShortDateString();
16    public string ToShortTimeString();
17    public string ToString(IFormatProvider provider);
18    public string ToString(string format, IFormatProvider provider);
19    public override string ToString();
20    public string ToString(string format);
21    public DateTime ToUniversalTime();
22    public DateTime Date { get; }
23    public int Day { get; }
24    public int DayOfYear { get; }
25    public int Hour { get; }
26    public int Millisecond { get; }
27    public int Minute { get; }
28    public int Month { get; }
29    public static DateTime Now { get; }
30    public int Second { get; }
31    public long Ticks { get; }
32    public TimeSpan TimeOfDay { get; }
33    public static DateTime Today { get; }
34    public static DateTime UtcNow { get; }
35    public int Year { get; }
36 }
37
38 // Namespace: System.Globalization, Library: BCL
39 public sealed class DateTimeFormatInfo: ICloneable, IFormatProvider
40 {
41     public DateTimeFormatInfo();
42     public object Clone();
43     public string GetAbbreviatedMonthName(int month);
44     public int GetEra(string eraName);

```

C# LANGUAGE SPECIFICATION

```
1     public string GetEraName(int era);
2     public object GetFormat(Type formatType);
3     public string GetMonthName(int month);
4     public static DateTimeFormatInfo ReadOnly(DateTimeFormatInfo dtfi);
5     public string[] AbbreviatedDayNames { get; set; }
6     public string[] AbbreviatedMonthNames { get; set; }
7     public string AMDesignator { get; set; }
8     public static DateTimeFormatInfo CurrentInfo { get; }
9     public string DateSeparator { get; set; }
10    public string[] DayNames { get; set; }
11    public string FullDateTimePattern { get; set; }
12    public static DateTimeFormatInfo InvariantInfo { get; }
13    public bool IsReadOnly { get; }
14    public string LongDatePattern { get; set; }
15    public string LongTimePattern { get; set; }
16    public string MonthDayPattern { get; set; }
17    public string[] MonthNames { get; set; }
18    public string PMDesignator { get; set; }
19    public string ShortDatePattern { get; set; }
20    public string ShortTimePattern { get; set; }
21    public string TimeSeparator { get; set; }
22    public string YearMonthPattern { get; set; }
23 }
24
25 // Namespace: System.Globalization, Library: BCL
26 public enum DateTimeStyles
27 {
28     AdjustToUniversal = 0x10,
29     AllowInnerWhite = 0x4,
30     AllowLeadingWhite = 0x1,
31     AllowTrailingWhite = 0x2,
32     AllowWhiteSpaces = AllowLeadingWhite | AllowTrailingWhite | AllowInnerWhite,
33     NoCurrentDateDefault = 0x8,
34     None = 0x0,
35 }
36
37 // Namespace: System, Library: ExtendedNumerics
38 public struct Decimal: IComparable, IFormattable
39 {
40     public Decimal(int value);
41     public Decimal(uint value);
42     public Decimal(long value);
43     public Decimal(ulong value);
44     public Decimal(float value);
```



```
1 public Decimal(double value);
2 public Decimal(int[] bits);
3 public static readonly decimal MaxValue;
4 public static readonly decimal MinusOne;
5 public static readonly decimal MinValue;
6 public static readonly decimal One;
7 public static readonly decimal Zero;
8 public static decimal Add(decimal d1, decimal d2);
9 public static int Compare(decimal d1, decimal d2);
10 public int CompareTo(object value);
11 public static decimal Divide(decimal d1, decimal d2);
12 public override bool Equals(object value);
13 public static bool Equals(decimal d1, decimal d2);
14 public static decimal Floor(decimal d);
15 public static int[] GetBits(decimal d);
16 public override int GetHashCode();
17 public static decimal Multiply(decimal d1, decimal d2);
18 public static decimal Negate(decimal d);
19 public static Decimal operator +(Decimal d1, Decimal d2);
20 public static Decimal operator --(Decimal d);
21 public static Decimal operator /(Decimal d1, Decimal d2);
22 public static bool operator ==(Decimal d1, Decimal d2);
23 public static explicit operator Decimal(float value);
24 public static explicit operator Decimal(double value);
25 public static explicit operator byte(Decimal value);
26 public static explicit operator sbyte(Decimal value);
27 public static explicit operator char(Decimal value);
28 public static explicit operator short(Decimal value);
29 public static explicit operator ushort(Decimal value);
30 public static explicit operator int(Decimal value);
31 public static explicit operator uint(Decimal value);
32 public static explicit operator long(Decimal value);
33 public static explicit operator ulong(Decimal value);
34 public static explicit operator float(Decimal value);
35 public static explicit operator double(Decimal value);
36 public static bool operator >(Decimal d1, Decimal d2);
37 public static bool operator >=(Decimal d1, Decimal d2);
38 public static implicit operator Decimal(byte value);
39 public static implicit operator Decimal(sbyte value);
40 public static implicit operator Decimal(short value);
41 public static implicit operator Decimal(ushort value);
42 public static implicit operator Decimal(char value);
43 public static implicit operator Decimal(int value);
44 public static implicit operator Decimal(uint value);
```

C# LANGUAGE SPECIFICATION

```
1     public static implicit operator Decimal(long value);
2     public static implicit operator Decimal(ulong value);
3     public static Decimal operator ++(Decimal d);
4     public static bool operator !=(Decimal d1, Decimal d2);
5     public static bool operator <(Decimal d1, Decimal d2);
6     public static bool operator <=(Decimal d1, Decimal d2);
7     public static Decimal operator %(Decimal d1, Decimal d2);
8     public static Decimal operator *(Decimal d1, Decimal d2);
9     public static Decimal operator -(Decimal d1, Decimal d2);
10    public static Decimal operator -(Decimal d);
11    public static Decimal operator +(Decimal d);
12    public static decimal Parse(string s);
13    public static decimal Parse(string s, NumberStyles style);
14    public static decimal Parse(string s, IFormatProvider provider);
15    public static decimal Parse(string s, NumberStyles style, IFormatProvider
16        provider);
17    public static decimal Remainder(decimal d1, decimal d2);
18    public static decimal Round(decimal d, int decimals);
19    public static decimal Subtract(decimal d1, decimal d2);
20    public string ToString(IFormatProvider provider);
21    public string ToString(string format, IFormatProvider provider);
22    public override string ToString();
23    public string ToString(string format);
24    public static decimal Truncate(decimal d);
25 }
26
27 // Namespace: System.Text, Library: BCL
28 public abstract class Decoder
29 {
30     protected Decoder();
31     public abstract int GetCharCount(byte[] bytes, int index, int count);
32     public abstract int GetChars(byte[] bytes, int byteIndex, int byteCount,
33         char[] chars, int charIndex);
34 }
35
36 // Namespace: System, Library: BCL
37 public abstract class Delegate: ICloneable
38 {
39     public virtual object Clone();
40     public static Delegate Combine(Delegate a, Delegate b);
41     public static Delegate Combine(Delegate[] delegates);
42     public override bool Equals(object obj);
43     public override int GetHashCode();
44     public virtual Delegate[] GetInvocationList();
```

```

1     public static bool operator ==(Delegate d1, Delegate d2);
2     public static bool operator !=(Delegate d1, Delegate d2);
3     public static Delegate Remove(Delegate source, Delegate value);
4     public static Delegate RemoveAll(Delegate source, Delegate value);
5     public object Target { get; }
6 }
7
8 // Namespace: System.Collections, Library: BCL
9 public struct DictionaryEntry
10 {
11     public DictionaryEntry(object key, object value);
12     public object Key { get; set; }
13     public object Value { get; set; }
14 }
15
16 // Namespace: System.IO, Library: BCL
17 public sealed class Directory
18 {
19     public static void Delete(string path);
20     public static void Delete(string path, bool recursive);
21     public static bool Exists(string path);
22     public static DateTime GetCreationTime(string path);
23     public static string GetCurrentDirectory();
24     public static string[] GetDirectories(string path);
25     public static string[] GetDirectories(string path, string searchPattern);
26     public static string GetDirectoryRoot(string path);
27     public static string[] GetFiles(string path);
28     public static string[] GetFiles(string path, string searchPattern);
29     public static string[] GetFileSystemEntries(string path);
30     public static string[] GetFileSystemEntries(string path, string
31         searchPattern);
32     public static DateTime GetLastAccessTime(string path);
33     public static DateTime GetLastWriteTime(string path);
34     public static void Move(string sourceDirName, string destDirName);
35     public static void SetCreationTime(string path, DateTime creationTime);
36     public static void SetCurrentDirectory(string path);
37     public static void SetLastAccessTime(string path, DateTime lastAccessTime);
38     public static void SetLastWriteTime(string path, DateTime lastWriteTime);
39 }
40
41 // Namespace: System.IO, Library: BCL
42 public class DirectoryNotFoundException: IOException
43 {
44     public DirectoryNotFoundException();

```

C# LANGUAGE SPECIFICATION

```
1     public DirectoryNotFoundException(string message);
2     public DirectoryNotFoundException(string message, Exception innerException);
3 }
4
5 // Namespace: System, Library: BCL
6 public class DivideByZeroException: ArithmeticException
7 {
8     public DivideByZeroException();
9     public DivideByZeroException(string message);
10    public DivideByZeroException(string message, Exception innerException);
11 }
12
13 // Namespace: System, Library: ExtendedNumerics
14 public struct Double: IComparable, IFormattable
15 {
16     public const double Epsilon = 4.9406564584124654e-324;
17     public const double MaxValue = 1.7976931348623157e+308;
18     public const double MinValue = -1.7976931348623157e+308;
19     public const double NaN = (double)0.0 / (double)0.0;
20     public const double NegativeInfinity = (double)-1.0 / (double)(0.0);
21     public const double PositiveInfinity = (double)1.0 / (double)(0.0);
22     public int CompareTo(object value);
23     public override bool Equals(object obj);
24     public override int GetHashCode();
25     public static bool IsInfinity(double d);
26     public static bool IsNaN(double d);
27     public static bool IsNegativeInfinity(double d);
28     public static bool IsPositiveInfinity(double d);
29     public static double Parse(string s);
30     public static double Parse(string s, NumberStyles style);
31     public static double Parse(string s, IFormatProvider provider);
32     public static double Parse(string s, NumberStyles style, IFormatProvider
33         provider);
34     public string ToString(IFormatProvider provider);
35     public string ToString(string format, IFormatProvider provider);
36     public override string ToString();
37     public string ToString(string format);
38 }
39
40 // Namespace: System, Library: BCL
41 public class DuplicateWaitObjectException: ArgumentException
42 {
43     public DuplicateWaitObjectException();
44     public DuplicateWaitObjectException(string parameterName);
```

```

1     public DuplicateWaitObjectException(string parameterName, string message);
2 }
3
4 // Namespace: System.Text, Library: BCL
5 public abstract class Encoder
6 {
7     protected Encoder();
8     public abstract int GetByteCount(char[] chars, int index, int count, bool
9         flush);
10    public abstract int GetBytes(char[] chars, int charIndex, int charCount,
11        byte[] bytes, int byteIndex, bool flush);
12 }
13
14 // Namespace: System.Text, Library: BCL
15 public abstract class Encoding
16 {
17     protected Encoding();
18     public static byte[] Convert(Encoding srcEncoding, Encoding dstEncoding,
19        byte[] bytes);
20     public static byte[] Convert(Encoding srcEncoding, Encoding dstEncoding,
21        byte[] bytes, int index, int count);
22     public override bool Equals(object value);
23     public abstract int GetByteCount(char[] chars, int index, int count);
24     public virtual int GetByteCount(string s);
25     public virtual int GetByteCount(char[] chars);
26     public virtual int GetBytes(string s, int charIndex, int charCount, byte[]
27        bytes, int byteIndex);
28     public virtual byte[] GetBytes(string s);
29     public abstract int GetBytes(char[] chars, int charIndex, int charCount,
30        byte[] bytes, int byteIndex);
31     public virtual byte[] GetBytes(char[] chars, int index, int count);
32     public virtual byte[] GetBytes(char[] chars);
33     public abstract int GetCharCount(byte[] bytes, int index, int count);
34     public virtual int GetCharCount(byte[] bytes);
35     public abstract int GetChars(byte[] bytes, int byteIndex, int byteCount,
36        char[] chars, int charIndex);
37     public virtual char[] GetChars(byte[] bytes, int index, int count);
38     public virtual char[] GetChars(byte[] bytes);
39     public virtual Decoder GetDecoder();
40     public virtual Encoder GetEncoder();
41     public override int GetHashCode();
42     public abstract int GetMaxByteCount(int charCount);
43     public abstract int GetMaxCharCount(int byteCount);
44     public virtual byte[] GetPreamble();

```

C# LANGUAGE SPECIFICATION

```
1     public virtual string GetString(byte[] bytes, int index, int count);
2     public virtual string GetString(byte[] bytes);
3     public static Encoding ASCII { get; }
4     public static Encoding BigEndianUnicode { get; }
5     public static Encoding Default { get; }
6     public static Encoding Unicode { get; }
7     public static Encoding UTF8 { get; }
8 }
9
10 // Namespace: System.IO, Library: BCL
11 public class EndOfStreamException: IOException
12 {
13     public EndOfStreamException();
14     public EndOfStreamException(string message);
15     public EndOfStreamException(string message, Exception innerException);
16 }
17
18 // Namespace: System, Library: BCL
19 public abstract class Enum: ValueType, IComparable, IFormattable
20 {
21     public int CompareTo(object target);
22     public override bool Equals(object obj);
23     public static string Format(Type enumType, object value, string format);
24     public override int GetHashCode();
25     public static string GetName(Type enumType, object value);
26     public static string[] GetNames(Type enumType);
27     public static Type GetUnderlyingType(Type enumType);
28     public static Array GetValues(Type enumType);
29     public static bool IsDefined(Type enumType, object value);
30     public static object Parse(Type enumType, string value);
31     public static object Parse(Type enumType, string value, bool ignoreCase);
32     public static object ToObject(Type enumType, object value);
33     public static object ToObject(Type enumType, sbyte value);
34     public static object ToObject(Type enumType, short value);
35     public static object ToObject(Type enumType, int value);
36     public static object ToObject(Type enumType, byte value);
37     public static object ToObject(Type enumType, ushort value);
38     public static object ToObject(Type enumType, uint value);
39     public static object ToObject(Type enumType, long value);
40     public static object ToObject(Type enumType, ulong value);
41     public string ToString(IFormatProvider provider);
42     public string ToString(string format, IFormatProvider provider);
43     public override string ToString();
44     public string ToString(string format);
```

```

1  }
2
3  // Namespace: System, Library: BCL
4  public sealed class Environment
5  {
6      public static void Exit(int exitCode);
7      public static string[] GetCommandLineArgs();
8      public static string GetEnvironmentVariable(string variable);
9      public static IDictionary GetEnvironmentVariables();
10     public static string CommandLine { get; }
11     public static int ExitCode { get; set; }
12     public bool HasShutdownStarted { get; }
13     public static string NewLine { get; }
14     public static string StackTrace { get; }
15     public static int TickCount { get; }
16     public static Version Version { get; }
17 }
18
19 // Namespace: System.Security.Permissions, Library: BCL
20 public sealed class EnvironmentPermission: CodeAccessPermission
21 {
22     public EnvironmentPermission(PermissionState state);
23     public EnvironmentPermission(EnvironmentPermissionAccess flag, string
24         pathList);
25     public override IPermission Copy();
26     public override void FromXml(SecurityElement esd);
27     public override IPermission Intersect(IPermission target);
28     public override bool IsSubsetOf(IPermission target);
29     public override SecurityElement ToXml();
30     public override IPermission Union(IPermission other);
31 }
32
33 // Namespace: System.Security.Permissions, Library: BCL
34 public enum EnvironmentPermissionAccess
35 {
36     AllAccess = Read | Write,
37     NoAccess = 0x0,
38     Read = 0x1,
39     Write = 0x2,
40 }
41
42 // Namespace: System.Security.Permissions, Library: BCL
43 public sealed class EnvironmentPermissionAttribute: CodeAccessSecurityAttribute
44 {

```

C# LANGUAGE SPECIFICATION

```
1     public EnvironmentPermissionAttribute(SecurityAction action);
2     public override IPermission CreatePermission();
3     public string All { get; }
4     public string Read { get; set; }
5     public string Write { get; set; }
6 }
7
8 // Namespace: System, Library: BCL
9 public class EventArgs
10 {
11     public EventArgs();
12     public static readonly EventArgs Empty;
13 }
14
15 // Namespace: System, Library: BCL
16 public delegate void EventHandler(object sender, EventArgs e);
17
18 // Namespace: System, Library: BCL
19 public class Exception
20 {
21     public Exception();
22     public Exception(string message);
23     public Exception(string message, Exception innerException);
24     public virtual Exception GetBaseException();
25     public override string ToString();
26     public Exception InnerException { get; }
27     public virtual string Message { get; }
28     public virtual string StackTrace { get; }
29 }
30
31 // Namespace: System, Library: BCL
32 public sealed class ExecutionEngineException: SystemException
33 {
34     public ExecutionEngineException();
35     public ExecutionEngineException(string message);
36     public ExecutionEngineException(string message, Exception innerException);
37 }
38
39 // Namespace: System.IO, Library: BCL
40 public sealed class File
41 {
42     public static StreamWriter AppendText(string path);
43     public static void Copy(string sourceFileName, string destFileName);
44     public static void Copy(string sourceFileName, string destFileName, bool
```



```

1     overwrite);
2     public static FileStream Create(string path);
3     public static FileStream Create(string path, int bufferSize);
4     public static StreamWriter CreateText(string path);
5     public static void Delete(string path);
6     public static bool Exists(string path);
7     public static DateTime GetCreationTime(string path);
8     public static DateTime GetLastAccessTime(string path);
9     public static DateTime GetLastWriteTime(string path);
10    public static void Move(string sourceFileName, string destFileName);
11    public static FileStream Open(string path, FileMode mode);
12    public static FileStream Open(string path, FileMode mode, FileAccess
13        access);
14    public static FileStream Open(string path, FileMode mode, FileAccess
15        access, FileShare share);
16    public static FileStream OpenRead(string path);
17    public static StreamReader OpenText(string path);
18    public static FileStream OpenWrite(string path);
19    public static void SetCreationTime(string path, DateTime creationTime);
20    public static void SetLastAccessTime(string path, DateTime lastAccessTime);
21    public static void SetLastWriteTime(string path, DateTime lastWriteTime);
22 }
23
24 // Namespace: System.IO, Library: BCL
25 public enum FileAccess
26 {
27     Read = 0x1,
28     Readwrite = Read | Write,
29     Write = 0x2,
30 }
31
32 // Namespace: System.Security.Permissions, Library: BCL
33 public sealed class FileIOPermission: CodeAccessPermission
34 {
35     public FileIOPermission(PermissionState state);
36     public FileIOPermission(FileIOPermissionAccess access, string path);
37     public override IPermission Copy();
38     public override void FromXml(SecurityElement esd);
39     public override IPermission Intersect(IPermission target);
40     public override bool IsSubsetOf(IPermission target);
41     public override SecurityElement ToXml();
42     public override IPermission Union(IPermission other);
43 }
44

```

C# LANGUAGE SPECIFICATION

```
1 // Namespace: System.Security.Permissions, Library: BCL
2 public enum FileIOPermissionAccess
3 {
4     AllAccess = Read | Write | Append | PathDiscovery,
5     Append = 0x4,
6     NoAccess = 0x0,
7     PathDiscovery = 0x8,
8     Read = 0x1,
9     Write = 0x2,
10 }
11
12 // Namespace: System.Security.Permissions, Library: BCL
13 public sealed class FileIOPermissionAttribute: CodeAccessSecurityAttribute
14 {
15     public FileIOPermissionAttribute(SecurityAction action);
16     public override IPermission CreatePermission();
17     public string All { get; set; }
18     public string Append { get; set; }
19     public string PathDiscovery { get; set; }
20     public string Read { get; set; }
21     public string Write { get; set; }
22 }
23
24 // Namespace: System.IO, Library: BCL
25 public class FileLoadException: IOException
26 {
27     public FileLoadException();
28     public FileLoadException(string message);
29     public FileLoadException(string message, Exception inner);
30     public FileLoadException(string message, string fileName);
31     public FileLoadException(string message, string fileName, Exception inner);
32     public override string ToString();
33     public string FileName { get; }
34     public override string Message { get; }
35 }
36
37 // Namespace: System.IO, Library: BCL
38 public enum FileMode
39 {
40     Append = 6,
41     Create = 2,
42     CreateNew = 1,
43     Open = 3,
44     OpenOrCreate = 4,
```

```

1     Truncate = 5,
2 }
3
4 // Namespace: System.IO, Library: BCL
5 public class FileNotFoundException: IOException
6 {
7     public FileNotFoundException();
8     public FileNotFoundException(string message);
9     public FileNotFoundException(string message, Exception innerException);
10    public FileNotFoundException(string message, string fileName);
11    public FileNotFoundException(string message, string fileName, Exception
12        innerException);
13    public override string ToString();
14    public string FileName { get; }
15    public override string Message { get; }
16 }
17
18 // Namespace: System.IO, Library: BCL
19 public enum FileShare
20 {
21     None = 0x0,
22     Read = 0x1,
23     Readwrite = Read | Write,
24     Write = 0x2,
25 }
26
27 // Namespace: System.IO, Library: BCL
28 public class FileStream: Stream
29 {
30     public FileStream(string path, FileMode mode);
31     public FileStream(string path, FileMode mode, FileAccess access);
32     public FileStream(string path, FileMode mode, FileAccess access, FileShare
33         share);
34     public FileStream(string path, FileMode mode, FileAccess access, FileShare
35         share, int bufferSize);
36     public FileStream(string path, FileMode mode, FileAccess access, FileShare
37         share, int bufferSize, bool useAsync);
38     public override IAsyncResult BeginRead(byte[] array, int offset, int
39         numBytes, AsyncCallback userCallback, object stateObject);
40     public override IAsyncResult BeginWrite(byte[] array, int offset, int
41         numBytes, AsyncCallback userCallback, object stateObject);
42     public override void Close();
43     protected virtual void Dispose(bool disposing);
44     public override int EndRead(IAsyncResult asyncResult);

```

C# LANGUAGE SPECIFICATION

```
1     public override void EndWrite(IAsyncResult asyncResult);
2     ~FileStream();
3     public override void Flush();
4     public override int Read(byte[] array, int offset, int count);
5     public override int ReadByte();
6     public override long Seek(long offset, SeekOrigin origin);
7     public override void SetLength(long value);
8     public override void Write(byte[] array, int offset, int count);
9     public override void WriteByte(byte value);
10    public override bool CanRead { get; }
11    public override bool CanSeek { get; }
12    public override bool CanWrite { get; }
13    public virtual bool IsAsync { get; }
14    public override long Length { get; }
15    public override long Position { get; set; }
16 }
17
18 // Namespace: System, Library: BCL
19 public class FlagsAttribute: Attribute
20 {
21     public FlagsAttribute();
22 }
23
24 // Namespace: System, Library: BCL
25 public class FormatException: SystemException
26 {
27     public FormatException();
28     public FormatException(string message);
29     public FormatException(string message, Exception innerException);
30 }
31
32 // Namespace: System, Library: BCL
33 public sealed class GC
34 {
35     public static void KeepAlive(object obj);
36     public static void ReRegisterForFinalize(object obj);
37     public static void SuppressFinalize(object obj);
38     public static void WaitForPendingFinalizers();
39 }
40
41 // Namespace: System.Collections, Library: BCL
42 public class Hashtable: ICloneable, ICollection, IDictionary, IEnumerable
43 {
44     public Hashtable();
```

```

1     public Hashtable(int capacity);
2     public Hashtable(IHashCodeProvider hcp, IComparer comparer);
3     public Hashtable(int capacity, IHashCodeProvider hcp, IComparer comparer);
4     public Hashtable(IDictionary d);
5     public Hashtable(IDictionary d, IHashCodeProvider hcp, IComparer comparer);
6     public virtual void Add(object key, object value);
7     public virtual void Clear();
8     public virtual object Clone();
9     public virtual bool Contains(object key);
10    public virtual bool ContainsKey(object key);
11    public virtual bool ContainsValue(object value);
12    public virtual void CopyTo(Array array, int arrayIndex);
13    public virtual IDictionaryEnumerator GetEnumerator();
14    protected virtual int GetHashCode(object key);
15    protected virtual bool KeyEquals(object item, object key);
16    public virtual void Remove(object key);
17    public static Hashtable Synchronized(Hashtable table);
18    IEnumerator IEnumerable.GetEnumerator();
19    int ICollection.Count { get; }
20    public virtual int Count { get; }
21    bool IDictionary.IsFixedSize { get; }
22    public virtual bool IsFixedSize { get; }
23    bool IDictionary.IsReadOnly { get; }
24    public virtual bool IsReadOnly { get; }
25    bool ICollection.IsSynchronized { get; }
26    public virtual bool IsSynchronized { get; }
27    public virtual object this[object key] { get; set; }
28    ICollection IDictionary.Keys { get; }
29    public virtual ICollection Keys { get; }
30    object ICollection.SyncRoot { get; }
31    public virtual object SyncRoot { get; }
32    ICollection IDictionary.Values { get; }
33    public virtual ICollection Values { get; }
34 }
35
36 // Namespace: System, Library: BCL
37 public interface IAsyncResult
38 {
39     object AsyncState { get; }
40     WaitHandle AsyncWaitHandle { get; }
41     bool CompletedSynchronously { get; }
42     bool IsCompleted { get; }
43 }
44

```

C# LANGUAGE SPECIFICATION

```
1 // Namespace: System, Library: BCL
2 public interface ICloneable
3 {
4     object Clone();
5 }
6
7 // Namespace: System.Collections, Library: BCL
8 public interface ICollection: IEnumerable
9 {
10     void CopyTo(Array array, int index);
11     int Count { get; }
12     bool IsSynchronized { get; }
13     object SyncRoot { get; }
14 }
15
16 // Namespace: System, Library: BCL
17 public interface IComparable
18 {
19     int CompareTo(object obj);
20 }
21
22 // Namespace: System.Collections, Library: BCL
23 public interface IComparer
24 {
25     int Compare(object x, object y);
26 }
27
28 // Namespace: System.Collections, Library: BCL
29 public interface IDictionary: ICollection, IEnumerable
30 {
31     void Add(object key, object value);
32     void Clear();
33     bool Contains(object key);
34     IDictionaryEnumerator GetEnumerator();
35     void Remove(object key);
36     bool IsFixedSize { get; }
37     bool IsReadOnly { get; }
38     object this[object key] { get; set; }
39     ICollection Keys { get; }
40     ICollection Values { get; }
41 }
42
43 // Namespace: System.Collections, Library: BCL
44 public interface IDictionaryEnumerator: IEnumerator
```

```
1  {
2      DictionaryEntry Entry { get; }
3      object Key { get; }
4      object Value { get; }
5  }
6
7  // Namespace: System, Library: BCL
8  public interface IDisposable
9  {
10     void Dispose();
11 }
12
13 // Namespace: System.Collections, Library: BCL
14 public interface IEnumerable
15 {
16     IEnumerator GetEnumerator();
17 }
18
19 // Namespace: System.Collections, Library: BCL
20 public interface IEnumerator
21 {
22     bool MoveNext();
23     void Reset();
24     object Current { get; }
25 }
26
27 // Namespace: System, Library: BCL
28 public interface IFormatProvider
29 {
30     object GetFormat(Type formatType);
31 }
32
33 // Namespace: System, Library: BCL
34 public interface IFormattable
35 {
36     string ToString(string format, IFormatProvider formatProvider);
37 }
38
39 // Namespace: System.Collections, Library: BCL
40 public interface IHashCodeProvider
41 {
42     int GetHashCode(object obj);
43 }
44
```

C# LANGUAGE SPECIFICATION

```
1 // Namespace: System.Collections, Library: BCL
2 public interface IList: ICollection, IEnumerable
3 {
4     int Add(object value);
5     void Clear();
6     bool Contains(object value);
7     int IndexOf(object value);
8     void Insert(int index, object value);
9     void Remove(object value);
10    void RemoveAt(int index);
11    bool IsFixedSize { get; }
12    bool IsReadOnly { get; }
13    object this[int index] { get; set; }
14 }
15
16 // Namespace: System, Library: BCL
17 public sealed class IndexOutOfRangeException: SystemException
18 {
19     public IndexOutOfRangeException();
20     public IndexOutOfRangeException(string message);
21     public IndexOutOfRangeException(string message, Exception innerException);
22 }
23
24 // Namespace: System, Library: BCL
25 public struct Int16: IComparable, IFormattable
26 {
27     public const short MaxValue = 32767;
28     public const short MinValue = -32768;
29     public int CompareTo(object value);
30     public override bool Equals(object obj);
31     public override int GetHashCode();
32     public static short Parse(string s);
33     public static short Parse(string s, NumberStyles style);
34     public static short Parse(string s, IFormatProvider provider);
35     public static short Parse(string s, NumberStyles style, IFormatProvider
36         provider);
37     public string ToString(IFormatProvider provider);
38     public string ToString(string format, IFormatProvider provider);
39     public override string ToString();
40     public string ToString(string format);
41 }
42
43 // Namespace: System, Library: BCL
44 public struct Int32: IComparable, IFormattable
```



```

1  {
2      public const int MaxValue = 2147483647;
3      public const int MinValue = -2147483648;
4      public int CompareTo(object value);
5      public override bool Equals(object obj);
6      public override int GetHashCode();
7      public static int Parse(string s);
8      public static int Parse(string s, NumberStyles style);
9      public static int Parse(string s, IFormatProvider provider);
10     public static int Parse(string s, NumberStyles style, IFormatProvider
11         provider);
12     public string ToString(IFormatProvider provider);
13     public string ToString(string format, IFormatProvider provider);
14     public override string ToString();
15     public string ToString(string format);
16 }
17
18 // Namespace: System, Library: BCL
19 public struct Int64: IComparable, IFormattable
20 {
21     public const long MaxValue = 9223372036854775807;
22     public const long MinValue = -9223372036854775808;
23     public int CompareTo(object value);
24     public override bool Equals(object obj);
25     public override int GetHashCode();
26     public static long Parse(string s);
27     public static long Parse(string s, NumberStyles style);
28     public static long Parse(string s, IFormatProvider provider);
29     public static long Parse(string s, NumberStyles style, IFormatProvider
30         provider);
31     public string ToString(IFormatProvider provider);
32     public string ToString(string format, IFormatProvider provider);
33     public override string ToString();
34     public string ToString(string format);
35 }
36
37 // Namespace: System.Threading, Library: BCL
38 public sealed class Interlocked
39 {
40     public static int CompareExchange(ref int location1, int value, int
41         comparand);
42     public static float CompareExchange(ref float location1, float value, float
43         comparand);
44     public static object CompareExchange(ref object location1, object value,

```

C# LANGUAGE SPECIFICATION

```
1     object comparand);
2     public static int Decrement(ref int location);
3     public static long Decrement(ref long location);
4     public static int Exchange(ref int location1, int value);
5     public static float Exchange(ref float location1, float value);
6     public static object Exchange(ref object location1, object value);
7     public static int Increment(ref int location);
8     public static long Increment(ref long location);
9 }
10
11 // Namespace: System, Library: BCL
12 public class InvalidCastException: SystemException
13 {
14     public InvalidCastException();
15     public InvalidCastException(string message);
16     public InvalidCastException(string message, Exception innerException);
17 }
18
19 // Namespace: System, Library: BCL
20 public class InvalidOperationException: SystemException
21 {
22     public InvalidOperationException();
23     public InvalidOperationException(string message);
24     public InvalidOperationException(string message, Exception innerException);
25 }
26
27 // Namespace: System, Library: BCL
28 public sealed class InvalidProgramException: SystemException
29 {
30     public InvalidProgramException();
31     public InvalidProgramException(string message);
32     public InvalidProgramException(string message, Exception inner);
33 }
34
35 // Namespace: System.IO, Library: BCL
36 public class IOException: SystemException
37 {
38     public IOException();
39     public IOException(string message);
40     public IOException(string message, Exception innerException);
41 }
42
43 // Namespace: System.Security, Library: BCL
44 public interface IPermission
```

```

1  {
2      IPermission Copy();
3      void Demand();
4      IPermission Intersect(IPermission target);
5      bool IsSubsetOf(IPermission target);
6      IPermission Union(IPermission target);
7  }
8
9  // Namespace: System, Library: BCL
10 public abstract class MarshalByRefObject
11 {
12 }
13
14 // Namespace: System, Library: ExtendedNumerics
15 public sealed class Math
16 {
17     public const double E = 2.71828182845905;
18     public const double PI = 3.14159265358979;
19     public static sbyte Abs(sbyte value);
20     public static short Abs(short value);
21     public static int Abs(int value);
22     public static long Abs(long value);
23     public static float Abs(float value);
24     public static double Abs(double value);
25     public static decimal Abs(decimal value);
26     public static double Acos(double d);
27     public static double Asin(double d);
28     public static double Atan(double d);
29     public static double Atan2(double y, double x);
30     public static long BigMul(int a, int b);
31     public static double Ceiling(double a);
32     public static double Cos(double d);
33     public static double Cosh(double value);
34     public static int DivRem(int a, int b, out int result);
35     public static long DivRem(long a, long b, out long result);
36     public static double Exp(double d);
37     public static double Floor(double d);
38     public static double IEEERemainder(double x, double y);
39     public static double Log(double d);
40     public static double Log(double a, double newBase);
41     public static double Log10(double d);
42     public static sbyte Max(sbyte val1, sbyte val2);
43     public static byte Max(byte val1, byte val2);
44     public static short Max(short val1, short val2);

```

C# LANGUAGE SPECIFICATION

```
1     public static ushort Max(ushort val1, ushort val2);
2     public static int Max(int val1, int val2);
3     public static uint Max(uint val1, uint val2);
4     public static long Max(long val1, long val2);
5     public static ulong Max(ulong val1, ulong val2);
6     public static float Max(float val1, float val2);
7     public static double Max(double val1, double val2);
8     public static decimal Max(decimal val1, decimal val2);
9     public static sbyte Min(sbyte val1, sbyte val2);
10    public static byte Min(byte val1, byte val2);
11    public static short Min(short val1, short val2);
12    public static ushort Min(ushort val1, ushort val2);
13    public static int Min(int val1, int val2);
14    public static uint Min(uint val1, uint val2);
15    public static long Min(long val1, long val2);
16    public static ulong Min(ulong val1, ulong val2);
17    public static float Min(float val1, float val2);
18    public static double Min(double val1, double val2);
19    public static decimal Min(decimal val1, decimal val2);
20    public static double Pow(double x, double y);
21    public static double Round(double a);
22    public static double Round(double value, int digits);
23    public static decimal Round(decimal d);
24    public static int Sign(sbyte value);
25    public static int Sign(short value);
26    public static int Sign(int value);
27    public static int Sign(long value);
28    public static int Sign(float value);
29    public static int Sign(double value);
30    public static int Sign(decimal value);
31    public static double Sin(double a);
32    public static double Sinh(double value);
33    public static double Sqrt(double d);
34    public static double Tan(double a);
35    public static double Tanh(double value);
36 }
37
38 // Namespace: System.IO, Library: BCL
39 public class MemoryStream: Stream
40 {
41     public MemoryStream();
42     public MemoryStream(int capacity);
43     public MemoryStream(byte[] buffer);
44     public MemoryStream(byte[] buffer, bool writable);
```

```

1     public MemoryStream(byte[] buffer, int index, int count);
2     public MemoryStream(byte[] buffer, int index, int count, bool writable);
3     public MemoryStream(byte[] buffer, int index, int count, bool writable,
4         bool publiclyVisible);
5     public override void Close();
6     public override void Flush();
7     public virtual byte[] GetBuffer();
8     public override int Read(byte[] buffer, int offset, int count);
9     public override int ReadByte();
10    public override long Seek(long offset, seekOrigin loc);
11    public override void SetLength(long value);
12    public virtual byte[] ToArray();
13    public override void Write(byte[] buffer, int offset, int count);
14    public override void WriteByte(byte value);
15    public virtual void WriteTo(Stream stream);
16    public override bool CanRead { get; }
17    public override bool CanSeek { get; }
18    public override bool CanWrite { get; }
19    public virtual int Capacity { get; set; }
20    public override long Length { get; }
21    public override long Position { get; set; }
22 }
23
24 // Namespace: System.Threading, Library: BCL
25 public sealed class Monitor
26 {
27     public static void Enter(object obj);
28     public static void Exit(object obj);
29     public static void Pulse(object obj);
30     public static void PulseAll(object obj);
31     public static bool TryEnter(object obj);
32     public static bool TryEnter(object obj, int millisecondsTimeout);
33     public static bool TryEnter(object obj, TimeSpan timeout);
34     public static bool Wait(object obj, int millisecondsTimeout);
35     public static bool Wait(object obj, TimeSpan timeout);
36     public static bool Wait(object obj);
37 }
38
39 // Namespace: System, Library: ExtendedNumerics
40 public class NotFiniteNumberException: ArithmeticException
41 {
42     public NotFiniteNumberException();
43     public NotFiniteNumberException(double offendingNumber);
44     public NotFiniteNumberException(string message);

```

C# LANGUAGE SPECIFICATION

```
1     public NotFiniteNumberException(string message, double offendingNumber);
2     public NotFiniteNumberException(string message, double offendingNumber,
3         Exception innerException);
4     public double OffendingNumber { get; }
5 }
6
7 // Namespace: System, Library: BCL
8 public class NotSupportedException: SystemException
9 {
10     public NotSupportedException();
11     public NotSupportedException(string message);
12     public NotSupportedException(string message, Exception innerException);
13 }
14
15 // Namespace: System, Library: BCL
16 public class NullReferenceException: SystemException
17 {
18     public NullReferenceException();
19     public NullReferenceException(string message);
20     public NullReferenceException(string message, Exception innerException);
21 }
22
23 // Namespace: System.Globalization, Library: BCL
24 public sealed class NumberFormatInfo: ICloneable, IFormatProvider
25 {
26     public NumberFormatInfo();
27     public object Clone();
28     public object GetFormat(Type formatType);
29     public static NumberFormatInfo ReadOnly(NumberFormatInfo nfi);
30     public int CurrencyDecimalDigits { get; set; }
31     public string CurrencyDecimalSeparator { get; set; }
32     public string CurrencyGroupSeparator { get; set; }
33     public int[] CurrencyGroupSizes { get; set; }
34     public int CurrencyNegativePattern { get; set; }
35     public int CurrencyPositivePattern { get; set; }
36     public string CurrencySymbol { get; set; }
37     public static NumberFormatInfo CurrentInfo { get; }
38     public static NumberFormatInfo InvariantInfo { get; }
39     public bool IsReadOnly { get; }
40     public string NaNSymbol { get; set; }
41     public string NegativeInfinitySymbol { get; set; }
42     public string NegativeSign { get; set; }
43     public int NumberDecimalDigits { get; set; }
44     public string NumberDecimalSeparator { get; set; }
```

```

1     public string NumberGroupSeparator { get; set; }
2     public int[] NumberGroupSizes { get; set; }
3     public int NumberNegativePattern { get; set; }
4     public int PercentDecimalDigits { get; set; }
5     public string PercentDecimalSeparator { get; set; }
6     public string PercentGroupSeparator { get; set; }
7     public int[] PercentGroupSizes { get; set; }
8     public int PercentNegativePattern { get; set; }
9     public int PercentPositivePattern { get; set; }
10    public string PercentSymbol { get; set; }
11    public string PerMilleSymbol { get; set; }
12    public string PositiveInfinitySymbol { get; set; }
13    public string PositiveSign { get; set; }
14 }
15
16 // Namespace: System.Globalization, Library: BCL
17 public enum NumberStyles
18 {
19     AllowCurrencySymbol = 0x100,
20     AllowDecimalPoint = 0x20,
21     AllowExponent = 0x80,
22     AllowHexSpecifier = 0x200,
23     AllowLeadingSign = 0x4,
24     AllowLeadingWhite = 0x1,
25     AllowParentheses = 0x10,
26     AllowThousands = 0x40,
27     AllowTrailingSign = 0x8,
28     AllowTrailingWhite = 0x2,
29     Any = AllowLeadingWhite | AllowTrailingWhite | AllowLeadingSign |
30         AllowTrailingSign | AllowParentheses | AllowDecimalPoint |
31         AllowThousands | AllowExponent | AllowCurrencySymbol,
32     Currency = AllowLeadingWhite | AllowTrailingWhite | AllowLeadingSign |
33         AllowTrailingSign | AllowParentheses | AllowDecimalPoint |
34         AllowThousands | AllowCurrencySymbol,
35     Float = AllowLeadingWhite | AllowTrailingWhite | AllowLeadingSign |
36         AllowDecimalPoint | AllowExponent,
37     HexNumber = AllowLeadingWhite | AllowTrailingWhite | AllowHexSpecifier,
38     Integer = AllowLeadingWhite | AllowTrailingWhite | AllowLeadingSign,
39     None = 0x0,
40     Number = AllowLeadingWhite | AllowTrailingWhite | AllowLeadingSign |
41         AllowTrailingSign | AllowDecimalPoint | AllowThousands,
42 }
43
44 // Namespace: System, Library: BCL

```

C# LANGUAGE SPECIFICATION

```
1 public class Object
2 {
3     public Object();
4     public virtual bool Equals(object obj);
5     public static bool Equals(object objA, object objB);
6     ~Object();
7     public virtual int GetHashCode();
8     public Type GetType();
9     protected object MemberwiseClone();
10    public static bool ReferenceEquals(object objA, object objB);
11    public virtual string ToString();
12 }
13
14 // Namespace: System, Library: BCL
15 public class ObjectDisposedException: InvalidOperationException
16 {
17     public ObjectDisposedException(string objectName);
18     public ObjectDisposedException(string objectName, string message);
19     public override string Message { get; }
20     public string ObjectName { get; }
21 }
22
23 // Namespace: System, Library: BCL
24 public sealed class ObsoleteAttribute: Attribute
25 {
26     public ObsoleteAttribute();
27     public ObsoleteAttribute(string message);
28     public ObsoleteAttribute(string message, bool error);
29     public bool IsError { get; }
30     public string Message { get; }
31 }
32
33 // Namespace: System, Library: BCL
34 public class OutOfMemoryException: SystemException
35 {
36     public OutOfMemoryException();
37     public OutOfMemoryException(string message);
38     public OutOfMemoryException(string message, Exception innerException);
39 }
40
41 // Namespace: System, Library: BCL
42 public class OverflowException: ArithmeticException
43 {
44     public OverflowException();
```



```

1     public OverflowException(string message);
2     public OverflowException(string message, Exception innerException);
3 }
4
5 // Namespace: System.IO, Library: BCL
6 public sealed class Path
7 {
8     public static readonly char AltDirectorySeparatorChar;
9     public static readonly char DirectorySeparatorChar;
10    public static readonly char PathSeparator;
11    public static string ChangeExtension(string path, string extension);
12    public static string Combine(string path1, string path2);
13    public static string GetDirectoryName(string path);
14    public static string GetExtension(string path);
15    public static string GetFileName(string path);
16    public static string GetFileNameWithoutExtension(string path);
17    public static string GetFullPath(string path);
18    public static string GetPathRoot(string path);
19    public static string GetTempFileName();
20    public static string GetTempPath();
21    public static bool HasExtension(string path);
22    public static bool IsPathRooted(string path);
23 }
24
25 // Namespace: System.IO, Library: BCL
26 public class PathTooLongException: IOException
27 {
28     public PathTooLongException();
29     public PathTooLongException(string message);
30     public PathTooLongException(string message, Exception innerException);
31 }
32
33 // Namespace: System.Security, Library: BCL
34 public class PermissionSet: ICollection, IEnumerable
35 {
36     public PermissionSet(PermissionState state);
37     public PermissionSet(PermissionSet permSet);
38     public virtual IPermission AddPermission(IPermission perm);
39     public virtual void Assert();
40     public virtual PermissionSet Copy();
41     public virtual void CopyTo(Array array, int index);
42     public virtual void Demand();
43     public virtual void Deny();
44     public virtual void FromXml(SecurityElement et);

```

C# LANGUAGE SPECIFICATION

```
1     public virtual IEnumerator GetEnumerator();
2     public virtual bool IsSubsetOf(PermissionSet target);
3     public virtual void PermitOnly();
4     public override string ToString();
5     public virtual SecurityElement ToXml();
6     public virtual PermissionSet Union(PermissionSet other);
7     int ICollection.Count { get; }
8     bool ICollection.IsSynchronized { get; }
9     object ICollection.SyncRoot { get; }
10  }
11
12  // Namespace: System.Security.Permissions, Library: BCL
13  public enum PermissionState
14  {
15      None = 0,
16      Unrestricted = 1,
17  }
18
19  // Namespace: System, Library: BCL
20  public class Random
21  {
22      public Random();
23      public Random(int Seed);
24      public virtual int Next(int maxValue);
25      public virtual int Next(int minValue, int maxValue);
26      public virtual int Next();
27      public virtual void NextBytes(byte[] buffer);
28      public virtual double NextDouble();
29  }
30
31  // Namespace: System, Library: BCL
32  public class RankException: SystemException
33  {
34      public RankException();
35      public RankException(string message);
36      public RankException(string message, Exception innerException);
37  }
38
39  // Namespace: System, Library: BCL
40  public struct SByte: IComparable, IFormattable
41  {
42      public const sbyte MaxValue = 127;
43      public const sbyte MinValue = -128;
44      public int CompareTo(object obj);
```

```

1     public override bool Equals(object obj);
2     public override int GetHashCode();
3     public static sbyte Parse(string s);
4     public static sbyte Parse(string s, NumberStyles style);
5     public static sbyte Parse(string s, IFormatProvider provider);
6     public static sbyte Parse(string s, NumberStyles style, IFormatProvider
7         provider);
8     public string ToString(IFormatProvider provider);
9     public string ToString(string format, IFormatProvider provider);
10    public override string ToString();
11    public string ToString(string format);
12 }
13
14 // Namespace: System.Security.Permissions, Library: BCL
15 public enum SecurityAction
16 {
17     Assert = 3,
18     Demand = 2,
19     Deny = 4,
20     InheritanceDemand = 7,
21     LinkDemand = 6,
22     PermitOnly = 5,
23     RequestMinimum = 8,
24     RequestOptional = 9,
25     RequestRefuse = 10,
26 }
27
28 // Namespace: System.Security.Permissions, Library: BCL
29 public abstract class SecurityAttribute: Attribute
30 {
31     protected SecurityAttribute();
32     public SecurityAttribute(SecurityAction action);
33     public abstract IPermission CreatePermission();
34     public bool Unrestricted { get; set; }
35 }
36
37 // Namespace: System.Security, Library: BCL
38 public sealed class SecurityElement
39 {
40     public override string ToString();
41 }
42
43 // Namespace: System.Security, Library: BCL
44 public class SecurityException: SystemException

```

C# LANGUAGE SPECIFICATION

```
1  {
2      public SecurityException();
3      public SecurityException(string message);
4      public SecurityException(string message, Exception inner);
5  }
6
7  // Namespace: System.Security.Permissions, Library: BCL
8  public sealed class SecurityPermission: CodeAccessPermission
9  {
10     public SecurityPermission(PermissionState state);
11     public SecurityPermission(SecurityPermissionFlag flag);
12     public override IPermission Copy();
13     public override void FromXml(SecurityElement esd);
14     public override IPermission Intersect(IPermission target);
15     public override bool IsSubsetOf(IPermission target);
16     public override SecurityElement ToXml();
17     public override IPermission Union(IPermission target);
18 }
19
20 // Namespace: System.Security.Permissions, Library: BCL
21 public sealed class SecurityPermissionAttribute: CodeAccessSecurityAttribute
22 {
23     public SecurityPermissionAttribute(SecurityAction action);
24     public override IPermission CreatePermission();
25     public SecurityPermissionFlag Flags { get; set; }
26 }
27
28 // Namespace: System.Security.Permissions, Library: BCL
29 public enum SecurityPermissionFlag
30 {
31     Assertion = 0x1,
32     ControlThread = 0x10,
33     Execution = 0x8,
34     NoFlags = 0x0,
35     SkipVerification = 0x4,
36     UnmanagedCode = 0x2,
37 }
38
39 // Namespace: System.IO, Library: BCL
40 public enum SeekOrigin
41 {
42     Begin = 0,
43     Current = 1,
44     End = 2,
```

```

1  }
2
3  // Namespace: System, Library: ExtendedNumerics
4  public struct Single: IComparable, IFormattable
5  {
6      public const float Epsilon = (float)1.401298E-45;
7      public const float MaxValue = (float)3.402823E+38;
8      public const float MinValue = (float)-3.402823E+38;
9      public const float NaN = (float)0.0 / (float)0.0;
10     public const float NegativeInfinity = (float)-1.0 / (float)0.0;
11     public const float PositiveInfinity = (float)1.0 / (float)0.0;
12     public int CompareTo(object value);
13     public override bool Equals(object obj);
14     public override int GetHashCode();
15     public static bool IsInfinity(float f);
16     public static bool IsNaN(float f);
17     public static bool IsNegativeInfinity(float f);
18     public static bool IsPositiveInfinity(float f);
19     public static float Parse(string s);
20     public static float Parse(string s, NumberStyles style);
21     public static float Parse(string s, IFormatProvider provider);
22     public static float Parse(string s, NumberStyles style, IFormatProvider
23         provider);
24     public string ToString(IFormatProvider provider);
25     public string ToString(string format, IFormatProvider provider);
26     public override string ToString();
27     public string ToString(string format);
28 }
29
30 // Namespace: System, Library: BCL
31 public sealed class StackOverflowException: SystemException
32 {
33     public StackOverflowException();
34     public StackOverflowException(string message);
35     public StackOverflowException(string message, Exception innerException);
36 }
37
38 // Namespace: System.IO, Library: BCL
39 public abstract class Stream: MarshalByRefObject, IDisposable
40 {
41     protected Stream();
42     public static readonly Stream Null;
43     public virtual IAsyncResult BeginRead(byte[] buffer, int offset, int count,
44         AsyncCallback callback, object state);

```

C# LANGUAGE SPECIFICATION

```
1     public virtual IAsyncResult BeginWrite(byte[] buffer, int offset, int
2         count, AsyncCallback callback, object state);
3     public virtual void Close();
4     protected virtual WaitHandle CreateWaitHandle();
5     public virtual int EndRead(IAsyncResult asyncResult);
6     public virtual void EndWrite(IAsyncResult asyncResult);
7     public abstract void Flush();
8     public abstract int Read(byte[] buffer, int offset, int count);
9     public virtual int ReadByte();
10    public abstract long Seek(long offset, SeekOrigin origin);
11    public abstract void SetLength(long value);
12    void IDisposable.Dispose();
13    public abstract void Write(byte[] buffer, int offset, int count);
14    public virtual void WriteByte(byte value);
15    public abstract bool CanRead { get; }
16    public abstract bool CanSeek { get; }
17    public abstract bool CanWrite { get; }
18    public abstract long Length { get; }
19    public abstract long Position { get; set; }
20 }
21
22 // Namespace: System.IO, Library: BCL
23 public class StreamReader: TextReader
24 {
25     public StreamReader(Stream stream);
26     public StreamReader(Stream stream, bool detectEncodingFromByteOrderMarks);
27     public StreamReader(Stream stream, Encoding encoding);
28     public StreamReader(Stream stream, Encoding encoding, bool
29         detectEncodingFromByteOrderMarks);
30     public StreamReader(Stream stream, Encoding encoding, bool
31         detectEncodingFromByteOrderMarks, int bufferSize);
32     public StreamReader(string path);
33     public StreamReader(string path, bool detectEncodingFromByteOrderMarks);
34     public StreamReader(string path, Encoding encoding);
35     public StreamReader(string path, Encoding encoding, bool
36         detectEncodingFromByteOrderMarks);
37     public StreamReader(string path, Encoding encoding, bool
38         detectEncodingFromByteOrderMarks, int bufferSize);
39     public override void Close();
40     public void DiscardBufferedData();
41     protected override void Dispose(bool disposing);
42     public override int Peek();
43     public override int Read(char[] buffer, int index, int count);
44     public override int Read();
```

```

1     public override string ReadLine();
2     public override string ReadToEnd();
3     public virtual Stream BaseStream { get; }
4     public virtual Encoding CurrentEncoding { get; }
5 }
6
7 // Namespace: System.IO, Library: BCL
8 public class StreamWriter: TextWriter
9 {
10    public StreamWriter(Stream stream);
11    public StreamWriter(Stream stream, Encoding encoding);
12    public StreamWriter(Stream stream, Encoding encoding, int bufferSize);
13    public StreamWriter(string path);
14    public StreamWriter(string path, bool append);
15    public StreamWriter(string path, bool append, Encoding encoding);
16    public StreamWriter(string path, bool append, Encoding encoding, int
17        bufferSize);
18    public override void Close();
19    protected override void Dispose(bool disposing);
20    ~StreamWriter();
21    public override void Flush();
22    public override void Write(string value);
23    public override void Write(char[] buffer, int index, int count);
24    public override void Write(char[] buffer);
25    public override void Write(char value);
26    public virtual bool AutoFlush { get; set; }
27    public virtual Stream BaseStream { get; }
28    public override Encoding Encoding { get; }
29 }
30
31 // Namespace: System, Library: BCL
32 public sealed class String: ICloneable, IComparable, IEnumerable
33 {
34    unsafe public String(char* value);
35    unsafe public String(char* value, int startIndex, int length);
36    public String(char[] value, int startIndex, int length);
37    public String(char[] value);
38    public String(char c, int count);
39    public static readonly string Empty;
40    public object Clone();
41    public static int Compare(string strA, string strB);
42    public static int Compare(string strA, string strB, bool ignoreCase);
43    public static int Compare(string strA, int indexA, string strB, int indexB,
44        int length);

```

C# LANGUAGE SPECIFICATION

```
1     public static int Compare(string strA, int indexA, string strB, int indexB,
2         int length, bool ignoreCase);
3     public static int CompareOrdinal(string strA, string strB);
4     public static int CompareOrdinal(string strA, int indexA, string strB, int
5         indexB, int length);
6     public int CompareTo(object value);
7     public static string Concat(object arg0, object arg1);
8     public static string Concat(object arg0, object arg1, object arg2);
9     public static string Concat(params object[] args);
10    public static string Concat(string str0, string str1);
11    public static string Concat(string str0, string str1, string str2);
12    public static string Concat(params string[] values);
13    public static string Copy(string str);
14    public void CopyTo(int sourceIndex, char[] destination, int
15        destinationIndex, int count);
16    public bool Endswith(string value);
17    public override bool Equals(object obj);
18    public static bool Equals(string a, string b);
19    public static string Format(string format, object arg0);
20    public static string Format(string format, object arg0, object arg1);
21    public static string Format(string format, object arg0, object arg1, object
22        arg2);
23    public static string Format(string format, params object[] args);
24    public static string Format(IFormatProvider provider, string format, params
25        object[] args);
26    public CharEnumerator GetEnumerator();
27    public override int GetHashCode();
28    public int IndexOf(char value);
29    public int IndexOf(char value, int startIndex);
30    public int IndexOf(char value, int startIndex, int count);
31    public int IndexOf(string value);
32    public int IndexOf(string value, int startIndex);
33    public int IndexOf(string value, int startIndex, int count);
34    public int IndexOfAny(char[] anyOf);
35    public int IndexOfAny(char[] anyOf, int startIndex);
36    public int IndexOfAny(char[] anyOf, int startIndex, int count);
37    public string Insert(int startIndex, string value);
38    public static string Intern(string str);
39    public static string IsInterned(string str);
40    public static string Join(string separator, string[] value);
41    public static string Join(string separator, string[] value, int startIndex,
42        int count);
43    public int LastIndexOf(char value);
44    public int LastIndexOf(char value, int startIndex);
```



```

1     public int LastIndexOf(char value, int startIndex, int count);
2     public int LastIndexOf(string value);
3     public int LastIndexOf(string value, int startIndex);
4     public int LastIndexOf(string value, int startIndex, int count);
5     public int LastIndexOfAny(char[] anyOf);
6     public int LastIndexOfAny(char[] anyOf, int startIndex);
7     public int LastIndexOfAny(char[] anyOf, int startIndex, int count);
8     public static bool operator ==(String a, String b);
9     public static bool operator !=(String a, String b);
10    public string PadLeft(int totalWidth);
11    public string PadLeft(int totalWidth, char paddingChar);
12    public string PadRight(int totalWidth);
13    public string PadRight(int totalWidth, char paddingChar);
14    public string Remove(int startIndex, int count);
15    public string Replace(char oldChar, char newChar);
16    public string Replace(string oldValue, string newValue);
17    public string[] Split(params char[] separator);
18    public string[] Split(char[] separator, int count);
19    public bool StartsWith(string value);
20    public string Substring(int startIndex);
21    public string Substring(int startIndex, int length);
22    IEnumerator IEnumerable.GetEnumerator();
23    public char[] ToCharArray();
24    public char[] ToCharArray(int startIndex, int length);
25    public string ToLower();
26    public string ToString(IFormatProvider provider);
27    public override string ToString();
28    public string ToUpper();
29    public string Trim(params char[] trimChars);
30    public string Trim();
31    public string TrimEnd(params char[] trimChars);
32    public string TrimStart(params char[] trimChars);
33    public char this[int index] { get; }
34    public int Length { get; }
35 }
36
37 // Namespace: System.Text, Library: BCL
38 public sealed class StringBuilder
39 {
40     public StringBuilder();
41     public StringBuilder(int capacity);
42     public StringBuilder(string value);
43     public StringBuilder Append(char value, int repeatCount);
44     public StringBuilder Append(char[] value, int startIndex, int charCount);

```

```
1     public StringBuilder Append(string value);
2     public StringBuilder Append(string value, int startIndex, int count);
3     public StringBuilder Append(bool value);
4     public StringBuilder Append(sbyte value);
5     public StringBuilder Append(byte value);
6     public StringBuilder Append(char value);
7     public StringBuilder Append(short value);
8     public StringBuilder Append(int value);
9     public StringBuilder Append(long value);
10    public StringBuilder Append(float value);
11    public StringBuilder Append(double value);
12    public StringBuilder Append(decimal value);
13    public StringBuilder Append(ushort value);
14    public StringBuilder Append(uint value);
15    public StringBuilder Append(ulong value);
16    public StringBuilder Append(object value);
17    public StringBuilder Append(char[] value);
18    public StringBuilder AppendFormat(string format, object arg0);
19    public StringBuilder AppendFormat(string format, object arg0, object arg1);
20    public StringBuilder AppendFormat(string format, object arg0, object arg1,
21        object arg2);
22    public StringBuilder AppendFormat(string format, params object[] args);
23    public StringBuilder AppendFormat(IFormatProvider provider, string format,
24        params object[] args);
25    public int EnsureCapacity(int capacity);
26    public bool Equals(StringBuilder sb);
27    public StringBuilder Insert(int index, string value, int count);
28    public StringBuilder Insert(int index, string value);
29    public StringBuilder Insert(int index, bool value);
30    public StringBuilder Insert(int index, sbyte value);
31    public StringBuilder Insert(int index, byte value);
32    public StringBuilder Insert(int index, short value);
33    public StringBuilder Insert(int index, char value);
34    public StringBuilder Insert(int index, char[] value);
35    public StringBuilder Insert(int index, char[] value, int startIndex, int
36        charCount);
37    public StringBuilder Insert(int index, int value);
38    public StringBuilder Insert(int index, long value);
39    public StringBuilder Insert(int index, float value);
40    public StringBuilder Insert(int index, double value);
41    public StringBuilder Insert(int index, decimal value);
42    public StringBuilder Insert(int index, ushort value);
43    public StringBuilder Insert(int index, uint value);
44    public StringBuilder Insert(int index, ulong value);
```

```

1     public StringBuilder Insert(int index, object value);
2     public StringBuilder Remove(int startIndex, int length);
3     public StringBuilder Replace(string oldValue, string newValue);
4     public StringBuilder Replace(string oldValue, string newValue, int
5         startIndex, int count);
6     public StringBuilder Replace(char oldChar, char newChar);
7     public StringBuilder Replace(char oldChar, char newChar, int startIndex,
8         int count);
9     public override string ToString();
10    public string ToString(int startIndex, int length);
11    public int Capacity { get; set; }
12    public char this[int index] { get; set; }
13    public int Length { get; set; }
14 }
15
16 // Namespace: System.IO, Library: BCL
17 public class StringReader: TextReader
18 {
19     public StringReader(string s);
20     public override void Close();
21     protected override void Dispose(bool disposing);
22     public override int Peek();
23     public override int Read(char[] buffer, int index, int count);
24     public override int Read();
25     public override string ReadLine();
26     public override string ReadToEnd();
27 }
28
29 // Namespace: System.IO, Library: BCL
30 public class StringWriter: TextWriter
31 {
32     public StringWriter();
33     public StringWriter(IFormatProvider formatProvider);
34     public StringWriter(StringBuilder sb);
35     public StringWriter(StringBuilder sb, IFormatProvider formatProvider);
36     public override void Close();
37     protected override void Dispose(bool disposing);
38     public virtual StringBuilder GetStringBuilder();
39     public override string ToString();
40     public override void write(string value);
41     public override void write(char[] buffer, int index, int count);
42     public override void write(char value);
43     public override Encoding Encoding { get; }
44 }

```

C# LANGUAGE SPECIFICATION

```
1
2 // Namespace: System.Threading, Library: BCL
3 public class SynchronizationLockException: SystemException
4 {
5     public SynchronizationLockException();
6     public SynchronizationLockException(string message);
7     public SynchronizationLockException(string message, Exception
8         innerException);
9 }
10
11 // Namespace: System, Library: BCL
12 public class SystemException: Exception
13 {
14     public SystemException();
15     public SystemException(string message);
16     public SystemException(string message, Exception innerException);
17 }
18
19 // Namespace: System.IO, Library: BCL
20 public abstract class TextReader: MarshalByRefObject, IDisposable
21 {
22     protected TextReader();
23     public static readonly TextReader Null;
24     public virtual void Close();
25     protected virtual void Dispose(bool disposing);
26     public virtual int Peek();
27     public virtual int Read(char[] buffer, int index, int count);
28     public virtual int Read();
29     public virtual int ReadBlock(char[] buffer, int index, int count);
30     public virtual string ReadLine();
31     public virtual string ReadToEnd();
32     public static TextReader Synchronized(TextReader reader);
33     void IDisposable.Dispose();
34 }
35
36 // Namespace: System.IO, Library: BCL
37 public abstract class TextWriter: MarshalByRefObject, IDisposable
38 {
39     protected TextWriter();
40     protected TextWriter(IFormatProvider formatProvider);
41     public static readonly TextWriter Null;
42     public virtual void Close();
43     protected virtual void Dispose(bool disposing);
44     public virtual void Flush();
```

```

1 public static TextWriter Synchronized(TextWriter writer);
2 void IDisposable.Dispose();
3 public virtual void Write(string format, params object[] arg);
4 public virtual void Write(string format, object arg0, object arg1, object
5     arg2);
6 public virtual void Write(string format, object arg0, object arg1);
7 public virtual void Write(string format, object arg0);
8 public virtual void Write(object value);
9 public virtual void Write(string value);
10 public virtual void Write(decimal value);
11 public virtual void Write(double value);
12 public virtual void Write(float value);
13 public virtual void Write(ulong value);
14 public virtual void Write(long value);
15 public virtual void Write(uint value);
16 public virtual void Write(int value);
17 public virtual void Write(bool value);
18 public virtual void Write(char[] buffer, int index, int count);
19 public virtual void Write(char[] buffer);
20 public virtual void Write(char value);
21 public virtual void WriteLine(string format, params object[] arg);
22 public virtual void WriteLine(string format, object arg0, object arg1,
23     object arg2);
24 public virtual void WriteLine(string format, object arg0, object arg1);
25 public virtual void WriteLine(string format, object arg0);
26 public virtual void WriteLine(object value);
27 public virtual void WriteLine(string value);
28 public virtual void WriteLine(decimal value);
29 public virtual void WriteLine(double value);
30 public virtual void WriteLine(float value);
31 public virtual void WriteLine(ulong value);
32 public virtual void WriteLine(long value);
33 public virtual void WriteLine(uint value);
34 public virtual void WriteLine(int value);
35 public virtual void WriteLine(bool value);
36 public virtual void WriteLine(char[] buffer, int index, int count);
37 public virtual void WriteLine(char[] buffer);
38 public virtual void WriteLine(char value);
39 public virtual void WriteLine();
40 public abstract Encoding Encoding { get; }
41 public virtual IFormatProvider FormatProvider { get; }
42 public virtual string NewLine { get; set; }
43 }
44

```

C# LANGUAGE SPECIFICATION

```
1 // Namespace: System, Library: BCL
2 public sealed class Thread
3 {
4     public Thread(ThreadStart start);
5     public void Abort(object stateInfo);
6     public void Abort();
7     ~Thread();
8     public void Join();
9     public bool Join(int millisecondsTimeout);
10    public bool Join(TimeSpan timeout);
11    public static void MemoryBarrier ();
12    public static void ResetAbort();
13    public static void Sleep(int millisecondsTimeout);
14    public static void Sleep(TimeSpan timeout);
15    public void Start();
16    public static byte VolatileRead (ref byte address);
17    public static short VolatileRead (ref short address);
18    public static int VolatileRead (ref int address);
19    public static long VolatileRead (ref long address);
20    public static sbyte VolatileRead (ref sbyte address);
21    public static ushort VolatileRead (ref ushort address);
22    public static uint VolatileRead (ref uint address);
23    public static ulong VolatileRead (ref ulong address);
24    public static float VolatileRead (ref float address);
25    public static double VolatileRead (ref double address);
26    public static object VolatileRead (ref object address);
27    public static void VolatileWrite (ref byte address, byte value);
28    public static void VolatileWrite (ref short address, short value);
29    public static void VolatileWrite (ref int address, int value);
30    public static void VolatileWrite (ref long address, long value);
31    public static void VolatileWrite (ref sbyte address, sbyte value);
32    public static void VolatileWrite (ref ushort address, ushort value);
33    public static void VolatileWrite (ref uint address, uint value);
34    public static void VolatileWrite (ref ulong address, ulong value);
35    public static void VolatileWrite (ref float address, float value);
36    public static void VolatileWrite (ref double address, double value);
37    public static void VolatileWrite (ref object address, object value);
38    public static Thread CurrentThread { get; }
39    public bool IsAlive { get; }
40    public bool IsBackground { get; set; }
41    public string Name { get; set; }
42    public ThreadPriority Priority { get; set; }
43    public ThreadState ThreadState { get; }
44 }
```

```
1
2 // Namespace: System.Threading, Library: BCL
3 public sealed class ThreadAbortException: SystemException
4 {
5     public object ExceptionState { get; }
6 }
7
8 // Namespace: System.Threading, Library: BCL
9 public enum ThreadPriority
10 {
11     AboveNormal = 3,
12     BelowNormal = 1,
13     Highest = 4,
14     Lowest = 0,
15     Normal = 2,
16 }
17
18 // Namespace: System.Threading, Library: BCL
19 public delegate void ThreadStart();
20
21 // Namespace: System.Threading, Library: BCL
22 public enum ThreadState
23 {
24     Aborted = 0x100,
25     AbortRequested = 0x80,
26     Background = 0x4,
27     Running = 0x0,
28     Stopped = 0x10,
29     Unstarted = 0x8,
30     WaitSleepJoin = 0x20,
31 }
32
33 // Namespace: System.Threading, Library: BCL
34 public class ThreadStateException: SystemException
35 {
36     public ThreadStateException();
37     public ThreadStateException(string message);
38     public ThreadStateException(string message, Exception innerException);
39 }
40
41 // Namespace: System.Threading, Library: BCL
42 public sealed class Timeout
43 {
44     public const int Infinite = -1;
```

C# LANGUAGE SPECIFICATION

```
1  }
2
3  // Namespace: System.Threading, Library: BCL
4  public sealed class Timer: MarshalByRefObject, IDisposable
5  {
6      public Timer(TimerCallback callback, object state, int dueTime, int period);
7      public Timer(TimerCallback callback, object state, TimeSpan dueTime,
8          TimeSpan period);
9      public bool Change(int dueTime, int period);
10     public bool Change(TimeSpan dueTime, TimeSpan period);
11     public void Dispose();
12     public bool Dispose(WaitHandle notifyObject);
13     ~Timer();
14 }
15
16 // Namespace: System.Threading, Library: BCL
17 public delegate void TimerCallback(object state);
18
19 // Namespace: System, Library: BCL
20 public struct TimeSpan: IComparable
21 {
22     public TimeSpan(long ticks);
23     public TimeSpan(int hours, int minutes, int seconds);
24     public TimeSpan(int days, int hours, int minutes, int seconds);
25     public TimeSpan(int days, int hours, int minutes, int seconds, int
26         milliseconds);
27     public static readonly TimeSpan MaxValue;
28     public static readonly TimeSpan MinValue;
29     public const long TicksPerDay = 864000000000;
30     public const long TicksPerHour = 36000000000;
31     public const long TicksPerMillisecond = 10000;
32     public const long TicksPerMinute = 600000000;
33     public const long TicksPerSecond = 10000000;
34     public static readonly TimeSpan Zero;
35     public TimeSpan Add(TimeSpan ts);
36     public static int Compare(TimeSpan t1, TimeSpan t2);
37     public int CompareTo(object value);
38     public TimeSpan Duration();
39     public override bool Equals(object value);
40     public static bool Equals(TimeSpan t1, TimeSpan t2);
41     public static TimeSpan FromDays(double value);
42     public static TimeSpan FromHours(double value);
43     public static TimeSpan FromMilliseconds(double value);
44     public static TimeSpan FromMinutes(double value);
```



```

1     public static TimeSpan FromSeconds(double value);
2     public static TimeSpan FromTicks(long value);
3     public override int GetHashCode();
4     public TimeSpan Negate();
5     public static TimeSpan operator +(TimeSpan t1, TimeSpan t2);
6     public static bool operator ==(TimeSpan t1, TimeSpan t2);
7     public static bool operator >(TimeSpan t1, TimeSpan t2);
8     public static bool operator >=(TimeSpan t1, TimeSpan t2);
9     public static bool operator !=(TimeSpan t1, TimeSpan t2);
10    public static bool operator <(TimeSpan t1, TimeSpan t2);
11    public static bool operator <=(TimeSpan t1, TimeSpan t2);
12    public static TimeSpan operator -(TimeSpan t1, TimeSpan t2);
13    public static TimeSpan operator -(TimeSpan t);
14    public static TimeSpan operator +(TimeSpan t);
15    public static TimeSpan Parse(string s);
16    public TimeSpan Subtract(TimeSpan ts);
17    public override string ToString();
18    public int Days { get; }
19    public int Hours { get; }
20    public int Milliseconds { get; }
21    public int Minutes { get; }
22    public int Seconds { get; }
23    public long Ticks { get; }
24    public double TotalDays { get; }
25    public double TotalHours { get; }
26    public double TotalMilliseconds { get; }
27    public double TotalMinutes { get; }
28    public double TotalSeconds { get; }
29 }
30
31 // Namespace: System, Library: BCL
32 public abstract class Type: Object
33 {
34     public virtual int GetArrayRank();
35     public abstract Type GetElementType();
36     public override int GetHashCode();
37     public virtual bool IsAssignableFrom(Type c);
38     public virtual bool IsInstanceOfType(object o);
39     public virtual bool IsSubclassOf(Type c);
40     public override string ToString();
41     public abstract Type BaseType { get; }
42     public abstract string FullName { get; }
43     public bool IsArray { get; }
44     public bool IsClass { get; }

```

C# LANGUAGE SPECIFICATION

```
1     public bool IsEnum { get; }
2     public bool IsInterface { get; }
3     public bool IsPointer { get; }
4     public bool IsValueType { get; }
5 }
6
7 // Namespace: System, Library: BCL
8 public sealed class TypeInitializationException: SystemException
9 {
10     public string TypeName { get; }
11 }
12
13 // Namespace: System, Library: BCL
14 public struct UInt16: IComparable, IFormattable
15 {
16     public const ushort MaxValue = 65535;
17     public const ushort MinValue = 0;
18     public int CompareTo(object value);
19     public override bool Equals(object obj);
20     public override int GetHashCode();
21     public static ushort Parse(string s);
22     public static ushort Parse(string s, NumberStyles style);
23     public static ushort Parse(string s, IFormatProvider provider);
24     public static ushort Parse(string s, NumberStyles style, IFormatProvider
25         provider);
26     public string ToString(IFormatProvider provider);
27     public string ToString(string format, IFormatProvider provider);
28     public override string ToString();
29     public string ToString(string format);
30 }
31
32 // Namespace: System, Library: BCL
33 public struct UInt32: IComparable, IFormattable
34 {
35     public const uint MaxValue = 4294967295;
36     public const uint MinValue = 0;
37     public int CompareTo(object value);
38     public override bool Equals(object obj);
39     public override int GetHashCode();
40     public static uint Parse(string s);
41     public static uint Parse(string s, NumberStyles style);
42     public static uint Parse(string s, IFormatProvider provider);
43     public static uint Parse(string s, NumberStyles style, IFormatProvider
44         provider);
```

```

1     public string ToString(IFormatProvider provider);
2     public string ToString(string format, IFormatProvider provider);
3     public override string ToString();
4     public string ToString(string format);
5 }
6
7 // Namespace: System, Library: BCL
8 public struct UInt64: IComparable, IFormattable
9 {
10    public const ulong MaxValue = 18446744073709551615;
11    public const ulong MinValue = 0;
12    public int CompareTo(object value);
13    public override bool Equals(object obj);
14    public override int GetHashCode();
15    public static ulong Parse(string s);
16    public static ulong Parse(string s, NumberStyles style);
17    public static ulong Parse(string s, IFormatProvider provider);
18    public static ulong Parse(string s, NumberStyles style, IFormatProvider
19        provider);
20    public string ToString(IFormatProvider provider);
21    public string ToString(string format, IFormatProvider provider);
22    public override string ToString();
23    public string ToString(string format);
24 }
25
26 // Namespace: System, Library: BCL
27 public class UnauthorizedAccessException: SystemException
28 {
29    public UnauthorizedAccessException();
30    public UnauthorizedAccessException(string message);
31    public UnauthorizedAccessException(string message, Exception inner);
32 }
33
34 // Namespace: System.Globalization, Library: BCL
35 public enum UnicodeCategory
36 {
37    ClosePunctuation = 21,
38    ConnectorPunctuation = 18,
39    Control = 14,
40    CurrencySymbol = 26,
41    DashPunctuation = 19,
42    DecimalDigitNumber = 8,
43    EnclosingMark = 7,
44    FinalQuotePunctuation = 23,

```

C# LANGUAGE SPECIFICATION

```
1     Format = 15,
2     InitialQuotePunctuation = 22,
3     LetterNumber = 9,
4     LineSeparator = 12,
5     LowercaseLetter = 1,
6     MathSymbol = 25,
7     ModifierLetter = 3,
8     ModifiersSymbol = 27,
9     NonSpacingMark = 5,
10    OpenPunctuation = 20,
11    OtherLetter = 4,
12    OtherNotAssigned = 29,
13    OtherNumber = 10,
14    OtherPunctuation = 24,
15    Othersymbol = 28,
16    ParagraphSeparator = 13,
17    PrivateUse = 17,
18    SpaceSeparator = 11,
19    SpacingCombiningMark = 6,
20    Surrogate = 16,
21    TitlecaseLetter = 2,
22    UppercaseLetter = 0,
23 }
24
25 // Namespace: System.Text, Library: BCL
26 public class UnicodeEncoding: Encoding
27 {
28     public UnicodeEncoding();
29     public UnicodeEncoding(bool bigEndian, bool byteOrderMark);
30     public override bool Equals(object value);
31     public override int GetByteCount(char[] chars, int index, int count);
32     public override int GetByteCount(string s);
33     public override int GetBytes(string s, int charIndex, int charCount, byte[]
34         bytes, int byteIndex);
35     public override byte[] GetBytes(string s);
36     public override int GetBytes(char[] chars, int charIndex, int charCount,
37         byte[] bytes, int byteIndex);
38     public override int GetCharCount(byte[] bytes, int index, int count);
39     public override int GetChars(byte[] bytes, int byteIndex, int byteCount,
40         char[] chars, int charIndex);
41     public override Decoder GetDecoder();
42     public override int GetHashCode();
43     public override int GetMaxByteCount(int charCount);
44     public override int GetMaxCharCount(int byteCount);
```

```
1     public override byte[] GetPreamble();
2 }
3
4 // Namespace: System.Text, Library: BCL
5 public class UTF8Encoding: Encoding
6 {
7     public UTF8Encoding();
8     public UTF8Encoding(bool encoderShouldEmitUTF8Identifier);
9     public UTF8Encoding(bool encoderShouldEmitUTF8Identifier, bool
10         throwOnInvalidBytes);
11     public override bool Equals(object value);
12     public override int GetByteCount(char[] chars, int index, int count);
13     public override int GetByteCount(string chars);
14     public override int GetBytes(string s, int charIndex, int charCount, byte[]
15         bytes, int byteIndex);
16     public override byte[] GetBytes(string s);
17     public override int GetBytes(char[] chars, int charIndex, int charCount,
18         byte[] bytes, int byteIndex);
19     public override int GetCharCount(byte[] bytes, int index, int count);
20     public override int GetChars(byte[] bytes, int byteIndex, int byteCount,
21         char[] chars, int charIndex);
22     public override Decoder GetDecoder();
23     public override Encoder GetEncoder();
24     public override int GetHashCode();
25     public override int GetMaxByteCount(int charCount);
26     public override int GetMaxCharCount(int byteCount);
27     public override byte[] GetPreamble();
28 }
29
30 // Namespace: System, Library: BCL
31 public abstract class ValueType
32 {
33     protected ValueType();
34     public override bool Equals(object obj);
35     public override int GetHashCode();
36     public override string ToString();
37 }
38
39 // Namespace: System.Security, Library: BCL
40 public class VerificationException: SystemException
41 {
42     public VerificationException();
43     public VerificationException(string message);
44     public VerificationException(string message, Exception innerException);
```

C# LANGUAGE SPECIFICATION

```
1  }
2
3  // Namespace: System, Library: BCL
4  public sealed class Version: ICloneable, IComparable
5  {
6      public Version(int major, int minor, int build, int revision);
7      public Version(int major, int minor, int build);
8      public Version(int major, int minor);
9      public Version(string version);
10     public Version();
11     public object Clone();
12     public int CompareTo(object version);
13     public override bool Equals(object obj);
14     public override int GetHashCode();
15     public static bool operator ==(Version v1, Version v2);
16     public static bool operator >(Version v1, Version v2);
17     public static bool operator >=(Version v1, Version v2);
18     public static bool operator !=(Version v1, Version v2);
19     public static bool operator <(Version v1, Version v2);
20     public static bool operator <=(Version v1, Version v2);
21     public int Build { get; }
22     public int Major { get; }
23     public int Minor { get; }
24     public int Revision { get; }
25 }
26
27 // Namespace: System.Threading, Library: BCL
28 public abstract class WaitHandle: MarshalByRefObject, IDisposable
29 {
30     public WaitHandle();
31     public virtual void Close();
32     protected virtual void Dispose(bool explicitDisposing);
33     ~WaitHandle();
34     void IDisposable.Dispose();
35     public static bool WaitAll(WaitHandle[] waitHandles);
36     public static int WaitAny(WaitHandle[] waitHandles);
37     public virtual bool WaitOne();
38 }
39
40 End of informative text.
```

E. Documentation Comments

This clause is informative.

C# provides a mechanism for programmers to document their code using a special comment syntax that contains XML text. Comments using such syntax are called *documentation comments*. The XML generation tool is called the *documentation generator*. (This generator could be, but need not be, the C# compiler itself.) The output produced by the documentation generator is called the *documentation file*. A documentation file is used as input to a *documentation viewer*; a tool intended to produce some sort of visual display of type information and its associated documentation.

A conforming C# compiler is not required to check the syntax of documentation comments; such comments are simply ordinary comments. A conforming compiler is permitted to do such checking, however.

This specification suggests a set of standard tags to be used in documentation comments. For C# implementations targeting the CLI, it also provides information about the documentation generator and the format of the documentation file. No information is provided about the documentation viewer.

E.1 Introduction

Comments having a special form can be used to direct a tool to produce XML from those comments and the source code elements, which they precede. Such comments are *single-line comments* of the form `///...` or *delimited comments* of the form `/** ... */`. They must immediately precede a user-defined type (such as a class, delegate, or interface) or a member (such as a field, event, property, or method) that they annotate.

Syntax:

```

single-line-doc-comment::
    ///input-charactersopt

delimited-doc-comment::
    /** delimited-comment-charactersopt */

```

In a *single-line-doc-comment*, if there is a *whitespace* character following the `///` characters on each of the *single-line-doc-comments* adjacent to the current *single-line-doc-comment*, then that *whitespace* character is not included in the XML output.

In a *delimited-doc-comment*, if the first non-*whitespace* character on the second line is an *asterisk* and the same pattern of optional *whitespace* characters and an *asterisk* character is repeated at the beginning of each of the lines within the *delimited-doc-comment*, then the characters of the repeated pattern are not included in the XML output. The pattern may include *whitespace* characters after, as well as before, the *asterisk* character.

Example:

```

/**
 * <remarks>Class <c>Point</c> models a point in a two-dimensional
 * plane.</remarks>
 */
public class Point
{
    ///method <c>draw</c> renders the point.</remarks>
    void draw() {...}
}

```

The text within documentation comments must be well formed according to the rules of XML (<http://www.w3.org/TR/REC-xml>). If the XML is ill formed, a warning is generated and the documentation file will contain a comment saying that an error was encountered.

1 Although developers are free to create their own set of tags, a recommended set is defined in §E.2. Some of
 2 the recommended tags have special meanings:

- 3 • The `<param>` tag is used to describe parameters. If such a tag is used, the documentation generator must
 4 verify that the specified parameter exists and that all parameters are described in documentation
 5 comments. If such verification fails, the documentation generator issues a warning.
- 6 • The `cref` attribute can be attached to any tag to provide a reference to a code element. The
 7 documentation generator must verify that this code element exists. If the verification fails, the
 8 documentation generator issues a warning. When looking for a name described in a `cref` attribute, the
 9 documentation generator must respect namespace visibility according to `using` statements appearing
 10 within the source code.
- 11 • The `<summary>` tag is intended to be used by a documentation viewer to display additional information
 12 about a type or member.

13 Note carefully that the documentation file does not provide full information about the type and members (for
 14 example, it does not contain any type information). To get such information about a type or member, the
 15 documentation file must be used in conjunction with reflection on the actual type or member.

16 **E.2 Recommended tags**

17 The documentation generator must accept and process any tag that is valid according to the rules of XML.
 18 The following tags provide commonly used functionality in user documentation. (Of course, other tags are
 19 possible.)

20

Tag	Reference	Purpose
<code><c></code>	§E.2.1	Set text in a code-like font
<code><code></code>	§E.2.2	Set one or more lines of source code or program output
<code><example></code>	§E.2.3	Indicate an example
<code><exception></code>	§E.2.4	Identifies the exceptions a method can throw
<code><list></code>	§E.2.5	Create a list or table
<code><para></code>	§E.2.6	Permit structure to be added to text
<code><param></code>	§E.2.7	Describe a parameter for a method or constructor
<code><paramref></code>	§E.2.8	Identify that a word is a parameter name
<code><permission></code>	§E.2.9	Document the security accessibility of a member
<code><remarks></code>	§E.2.10	Describe a type
<code><returns></code>	§E.2.11	Describe the return value of a method
<code><see></code>	§E.2.12	Specify a link
<code><seealso></code>	§E.2.13	Generate a <i>See Also</i> entry
<code><summary></code>	§E.2.14	Describe a member of a type
<code><value></code>	§E.2.15	Describe a property

21

22 **E.2.1 <c>**

23 This tag provides a mechanism to indicate that a fragment of text within a description should be set a special
 24 font such as that used for a block of code. (For lines of actual code, use `<code>` (§E.2.2).)

Syntax:

```
1      <c>text to be set like code</c>
```

Example:

```
4      ///  
5      ///  
6      public class Point  
7      {  
8          // ...  
9      }
```

E.2.2 <code>

11 This tag is used to set one or more lines of source code or program output in some special font. (For small
12 code fragments in narrative, use <c> (§E.2.1).)

Syntax:

```
14      <code>source code or program output</code>
```

Example:

```
16      ///  
17      ///  
18      ///  
19      ///  
20      ///  
21      ///  
22      ///  
23      ///  
24      ///  
25      ///  
26      public void Translate(int xor, int yor) {  
27          X += xor;  
28          Y += yor;  
29      }
```

E.2.3 <example>

31 This tag allows example code within a comment, to specify how a method or other library member may be
32 used. Ordinarily, this would also involve use of the tag <code> (§E.2.2) as well.

Syntax:

```
34      <example>description</example>
```

Example:

36 See <code> (§E.2.2) for an example.

E.2.4 <exception>

38 This tag provides a way to document the exceptions a method can throw.

Syntax:

```
40      <exception cref="member">description</exception>
```

41 where

```
42      cref="member"
```

43 The name of a member. The documentation generator checks that the given member exists and
44 translates *member* to the canonical element name in the documentation file.

```
45      description
```

46 A description of the circumstances in which the exception is thrown.

Example:

```

1      public class DataBaseOperations
2      {
3          /// <exception cref="MasterFileFormatCorruptException"></exception>
4          /// <exception cref="MasterFileLockedOpenException"></exception>
5          public static void ReadRecord(int flag) {
6              if (flag == 1)
7                  throw new MasterFileFormatCorruptException();
8              else if (flag == 2)
9                  throw new MasterFileLockedOpenException();
10             // ...
11         }
12     }

```

13 E.2.5 <list>

14 This tag is used to create a list or table of items. It may contain a <listheader> block to define the
15 heading row of either a table or definition list. (When defining a table, only an entry for *term* in the heading
16 need be supplied.)

17 Each item in the list is specified with an <item> block. When creating a definition list, both *term* and
18 *description* must be specified. However, for a table, bulleted list, or numbered list, only *description*
19 need be specified.

20 Syntax:

```

21     <list type="bullet" | "number" | "table">
22         <listheader>
23             <term>term</term>
24             <description>description</description>
25         </listheader>
26         <item>
27             <term>term</term>
28             <description>description</description>
29         </item>
30         ...
31         <item>
32             <term>term</term>
33             <description>description</description>
34         </item>
35     </list>

```

36 where

37 *term*

38 The term to define, whose definition is in *description*.

39 *description*

40

41 Either an item in a bullet or numbered list, or the definition of a *term*.

42 Example:

```

43     public class MyClass
44     {
45         /// <remarks>Here is an example of a bulleted list:
46         /// <list type="bullet">
47         /// <item>
48         /// <description>Item 1.</description>
49         /// </item>
50         /// <item>
51         /// <description>Item 2.</description>
52         /// </item>
53         /// </list>
54         /// </remarks>
55         public static void Main () {
56             // ...
57         }
58     }

```

E.2.6 <para>

This tag is for use inside other tags, such as <remarks> (§E.2.10) or <returns> (§E.2.11), and permits structure to be added to text.

Syntax:

```
<para>content</para>
```

where

content

The text of the paragraph.

Example:

```

10     /// <summary>This is the entry point of the Point class testing
11     program.
12     /// <para>This program tests each method and operator, and
13     /// is intended to be run after any non-trivial maintenance has
14     /// been performed on the Point class.</para></summary>
15     public static void Main() {
16         // ...
17     }
```

E.2.7 <param>

This tag is used to describe a parameter for a method, constructor, or indexer.

Syntax:

```
<param name="name">description</param>
```

where

name

The name of the parameter.

description

A description of the parameter.

Example:

```

28     /// <summary>This method changes the point's location to
29     /// the given coordinates.</summary>
30     /// <param><c>xor</c> is the new x-coordinate.</param>
31     /// <param><c>yor</c> is the new y-coordinate.</param>
32     public void Move(int xor, int yor) {
33         X = xor;
34         Y = yor;
35     }
```

E.2.8 <paramref>

This tag is used to indicate that a word is a parameter. The documentation file can be processed to format this parameter in some distinct way.

Syntax:

```
<paramref name="name" />
```

where

name

The name of the parameter.

Example:

```

1      /// <summary>This constructor initializes the new Point to
2      ///     (<paramref name="xor"/>,<paramref name="yor"/>).</summary>
3      /// <param><c>xor</c> is the new Point's x-coordinate.</param>
4      /// <param><c>yor</c> is the new Point's y-coordinate.</param>
5      public Point(int xor, int yor) {
6          X = xor;
7          Y = yor;
8      }

```

9 **E.2.9 <permission>**

10 This tag allows the security accessibility of a member to be documented.

11 **Syntax:**

```
12     <permission cref="member">description</permission>
```

13 where

```
14     cref="member"
```

15 The name of a member. The documentation generator checks that the given code element exists
16 and translates *member* to the canonical element name in the documentation file.

```
17     description
```

18 A description of the access to the member.

19 **Example:**

```

20     /// <permission cref="System.Security.PermissionSet">Everyone can
21     /// access this method.</permission>
22     public static void Test() {
23         // ...
24     }

```

25 **E.2.10 <remarks>**

26 This tag is used to specify overview information about a type. (Use <summary> (§E.2.14) to describe the
27 members of a type.)

28 **Syntax:**

```
29     <remarks>description</remarks>
```

30 where

```
31     description
```

32 The text of the remarks.

33 **Example:**

```

34     /// <remarks>Class <c>Point</c> models a point in a two-dimensional
35     /// plane.</remarks>
36     public class Point
37     {
38         // ...
39     }

```

40 **E.2.11 <returns>**

41 This tag is used to describe the return value of a method.

42 **Syntax:**

```
43     <returns>description</returns>
```

44 where

```
45     description
```

46 A description of the return value.

1 Example:

```

2      /// <summary>Report a point's location as a string.</summary>
3      /// <returns>A string representing a point's location, in the form
4      (x,y),
5      /// without any leading, training, or embedded whitespace.</returns>
6      public override string ToString() {
7          return "(" + X + "," + Y + ")";
8      }

```

9 E.2.12 <see>

10 This tag allows a link to be specified within text. (Use <seealso> (§E.2.13) to indicate text that is to appear
 11 in a *See Also* section.)

12 Syntax:

```
13      <see cref="member" />
```

14 where

```
15      cref="member"
```

16 The name of a member. The documentation generator checks that the given code element exists
 17 and passes *member* to the element name in the documentation file.

18 Example:

```

19      /// <summary>This method changes the point's location to
20      /// the given coordinates.</summary>
21      /// <see cref="Translate" />
22      public void Move(int xor, int yor) {
23          X = xor;
24          Y = yor;
25      }
26
27      /// <summary>This method changes the point's location by
28      /// the given x- and y-offsets.
29      /// </summary>
30      /// <see cref="Move" />
31      public void Translate(int xor, int yor) {
32          X += xor;
33          Y += yor;
34      }

```

34 E.2.13 <seealso>

35 This tag allows an entry to be generated for the *See Also* section. (Use <see> (§E.2.12) to specify a link
 36 from within text.)

37 Syntax:

```
38      <seealso cref="member" />
```

39 where

```
40      cref="member"
```

41 The name of a member. The documentation generator checks that the given code element exists
 42 and passes *member* to the element name in the documentation file.

43 Example:

```

44      /// <summary>This method determines whether two Points have the same
45      /// location.</summary>
46      /// <seealso cref="operator==" />
47      /// <seealso cref="operator!=" />
48      public override bool Equals(object o) {
49          // ...
50      }

```

1 **E.2.14 <summary>**

2 This tag can be used to describe a member for a type. (Use <remarks> (§E.2.10) to describe the type itself.)

3 **Syntax:**

4 `<summary>description</summary>`

5 where

6 *description*

7 A summary of the member.

8 **Example:**

```
9 /// <summary>This constructor initializes the new Point to  
10 (0,0).</summary>  
11 public Point() : this(0,0) {  
12 }
```

13 **E.2.15 <value>**

14 This tag allows a property to be described.

15 **Syntax:**

16 `<value>property description</value>`

17 where

18 *property description*

19 A description for the property.

20 **Example:**

```
21 /// <value>Property <c>X</c> represents the point's x-coordinate.</value>  
22 public int X  
23 {  
24     get { return x; }  
25     set { x = value; }  
26 }
```

27 **E.3 Processing the documentation file**

28 The following information is intended for C# implementations targeting the CLI.

29 The documentation generator generates an ID string for each element in the source code that is tagged with a
30 documentation comment. This ID string uniquely identifies a source element. A documentation viewer can
31 use an ID string to identify the corresponding metadata/reflection item to which the documentation applies.

32 The documentation file is not a hierarchical representation of the source code; rather, it is a flat list with a
33 generated ID string for each element.

34 **E.3.1 ID string format**

35 The documentation generator observes the following rules when it generates the ID strings:

- 36 • No white space is placed in the string.
- 37 • The first part of the string identifies the kind of member being documented, via a single character
38 followed by a colon. The following kinds of members are defined:

Character	Description
E	Event
F	Field
M	Method (including constructors, destructors, and operators)
N	Namespace
P	Property (including indexers)
T	Type (such as class, delegate, enum, interface, and struct)
!	Error string; the rest of the string provides information about the error. For example, the documentation generator generates error information for links that cannot be resolved.

- 1 • The second part of the string is the fully qualified name of the element, starting at the root of the
2 namespace. The name of the element, its enclosing type(s), and namespace are separated by periods. If
3 the name of the item itself has periods, they are replaced by the NUMBER SIGN # (U+000D). (It is
4 assumed that no element has this character in its name.)
- 5 • For methods and properties with arguments, the argument list follows, enclosed in parentheses. For
6 those without arguments, the parentheses are omitted. The arguments are separated by commas. The
7 encoding of each argument is the same as a CLI signature, as follows: Arguments are represented by
8 their fully qualified name. For example, `int` becomes `System.Int32`, `string` becomes
9 `System.String`, `object` becomes `System.Object`, and so on. Arguments having the `out` or `ref`
10 modifier have a '@' following their type name. Arguments passed by value or via `params` have no
11 special notation. Arguments that are arrays are represented as `[lowerbound:size, ..., lowerbound:size]`
12 where the number of commas is the rank - 1, and the lower bounds and size of each dimension, if
13 known, are represented in decimal. If a lower bound or size is not specified, it is omitted. If the lower
14 bound and size for a particular dimension are omitted, the ':' is omitted as well. Jagged arrays are
15 represented by one "[]" per level. Arguments that have pointer types other than void are represented
16 using a '*' following the type name. A void pointer is represented using a type name of
17 `"System.Void"`.

18 E.3.2 ID string examples

19 The following examples each show a fragment of C# code, along with the ID string produced from each
20 source element capable of having a documentation comment:

- 21 • Types are represented using their fully qualified name.

```

22     enum Color {Red, Blue, Green};
23     namespace Acme
24     {
25     interface IProcess { /* ... */ }
26     struct ValueType { /* ... */ }
27
28     class widget: Iprocess
29     {
30     public class NestedClass { /* ... */ }
31     public interface IMenuItem { /* ... */ }
32     public delegate void Del(int i);
33     public enum Direction {North, South, East, West};
34     }

```

C# LANGUAGE SPECIFICATION

```
1      "T:Color"
2      "T:Acme.IProcess"
3      "T:Acme.ValueType"
4      "T:Acme.Widget"
5      "T:Acme.Widget.NestedClass"
6      "T:Acme.Widget.IMenuItem"
7      "T:Acme.Widget.De1"
8      "T:Acme.Widget.Direction"
```

- 9 • Fields are represented by their fully qualified name.

```
10     namespace Acme
11     {
12     struct ValueType
13     {
14         private int total;
15     }
16     class widget: Iprocess
17     {
18         public class NestedClass
19         {
20             private int value;
21         }
22         private string message;
23         private static Color defaultColor;
24         private const double PI = 3.14159;
25         protected readonly double monthlyAverage;
26         private long[] array1;
27         private widget[,] array2;
28         private unsafe int *pCount;
29         private unsafe float **ppValues;
30     }
31 }
32 "F:Acme.ValueType.total"
33 "F:Acme.Widget.NestedClass.value"
34 "F:Acme.widget.message"
35 "F:Acme.widget.defaultColor"
36 "F:Acme.Widget.PI"
37 "F:Acme.widget.monthlyAverage"
38 "F:Acme.Widget.array1"
39 "F:Acme.Widget.array2"
40 "F:Acme.widget.pCount"
41 "F:Acme.Widget.ppValues"
```

- 42 • Constructors.

```
43     namespace Acme
44     {
45     class widget: Iprocess
46     {
47         static widget() { /* ... */ }
48         public widget() { /* ... */ }
49         public widget(string s) { /* ... */ }
50     }
51 }
52 "M:Acme.widget.#cctor"
53 "M:Acme.Widget.#ctor"
54 "M:Acme.Widget.#ctor(System.String)"
```

- 55 • Destructors.

```
56     namespace Acme
57     {
58     class widget: Iprocess
59     {
60         ~widget() { /* ... */ }
61     }
62 }
```



```

1      "M:Acme.Widget.Finalize"
2  • Methods.
3      namespace Acme
4      {
5      struct ValueType
6      {
7          public void M(int i) { /* ... */ }
8      }
9      class widget: Iprocess
10     {
11         public class NestedClass
12         {
13             public void M(int i) { /* ... */ }
14         }
15         public static void M0() { /* ... */ }
16         public void M1(char c, out float f, ref valueType v) { /* ... */ }
17         public void M2(short[] x1, int[,] x2, long[][] x3) { /* ... */ }
18         public void M3(long[][] x3, widget[,] x4) { /* ... */ }
19         public unsafe void M4(char *pc, Color **pf) { /* ... */ }
20         public unsafe void M5(void *pv, double *[,] pd) { /* ... */ }
21         public void M6(int i, params object[] args) { /* ... */ }
22     }
23
24     "M:Acme.ValueType.M(System.Int32)"
25     "M:Acme.Widget.NestedClass.M(System.Int32)"
26     "M:Acme.Widget.M0"
27     "M:Acme.Widget.M1(System.Char,System.Single@,Acme.ValueType@)"
28     "M:Acme.Widget.M2(System.Int16[],System.Int32[0:,0:],System.Int64[][])"
29     "M:Acme.Widget.M3(System.Int64[][] ,Acme.Widget[0:,0:,0:][])"
30     "M:Acme.Widget.M4(System.Char*,Color**)"
31     "M:Acme.Widget.M5(=VOID*,System.Double*[0:,0:][])"
32     "M:Acme.Widget.M6(System.Int32,System.Object[])"
33
34  • Properties and indexers.
35     namespace Acme
36     {
37     class widget: Iprocess
38     {
39         public int width {get { /* ... */ } set { /* ... */ }}
40         public int this[int i] {get { /* ... */ } set { /* ... */ }}
41         public int this[string s, int i] {get { /* ... */ } set { /* ... */ }}
42     }
43
44     "P:Acme.Widget.Width"
45     "P:Acme.Widget.Item(System.Int32)"
46     "P:Acme.Widget.Item(System.String,System.Int32)"
47  • Events
48     namespace Acme
49     {
50     class widget: Iprocess
51     {
52         public event Del AnEvent;
53     }
54
55     "E:Acme.Widget.AnEvent"
56  • Unary operators.

```

```

1      namespace Acme
2      {
3      class widget: Iprocess
4      {
5          public static widget operator+(widget x) { /* ... */ }
6      }
7      }
8
9      "M:Acme.Widget.op_UnaryPlus(Acme.Widget)"
10
11     The complete set of unary operator function names used is as follows: op_UnaryPlus,
12     op_UnaryNegation, op_Negation, op_OnesComplement, op_Increment, op_Decrement,
13     op_True, and op_False.

```

- Binary operators.

```

13     namespace Acme
14     {
15     class widget: Iprocess
16     {
17         public static widget operator+(widget x1, widget x2) { return x1; }
18     }
19     }
20
21     "M:Acme.Widget.op_Addition(Acme.Widget,Acme.Widget)"
22
23     The complete set of binary operator function names used is as follows: op_Addition,
24     op_Subtraction, op_Multiply, op_Division, op_Modulus, op_BitwiseAnd,
25     op_BitwiseOr, op_ExclusiveOr, op_LeftShift, op_RightShift, op_Equality,
26     op_Inequality, op_LessThan, op_LessThanOrEqual, op_GreaterThan, and
27     op_GreaterThanOrEqual.

```

- Conversion operators have a trailing '~' followed by the return type.

```

27     namespace Acme
28     {
29     class widget: Iprocess
30     {
31         public static explicit operator int(widget x) { /* ... */ }
32         public static implicit operator long(widget x) { /* ... */ }
33     }
34     }
35
36     "M:Acme.Widget.op_Explicit(Acme.Widget)~System.Int32"
37     "M:Acme.Widget.op_Implicit(Acme.Widget)~System.Int64"

```

37 E.4 An example

38 E.4.1 C# source code

39 The following example shows the source code of a Point class:

```

40     namespace Graphics
41     {
42
43         /// <remarks>Class <c>Point</c> models a point in a two-dimensional
44         plane.
45         /// </remarks>
46         public class Point
47         {
48
49             /// <summary>Instance variable <c>x</c> represents the point's
50             /// x-coordinate.</summary>
51             private int x;
52
53             /// <summary>Instance variable <c>y</c> represents the point's
54             /// y-coordinate.</summary>
55             private int y;

```

```

1      /// <value>Property <c>X</c> represents the point's x-
2      coordinate.</value>
3      public int X
4      {
5          get { return x; }
6          set { x = value; }
7      }
8
9      /// <value>Property <c>Y</c> represents the point's y-
10     coordinate.</value>
11     public int Y
12     {
13         get { return y; }
14         set { y = value; }
15     }
16
17     /// <summary>This constructor initializes the new Point to
18     /// (0,0).</summary>
19     public Point() : this(0,0) {}
20
21     /// <summary>This constructor initializes the new Point to
22     /// (<paramref name="xor"/>,<paramref name="yor"/>).</summary>
23     /// <param><c>xor</c> is the new Point's x-coordinate.</param>
24     /// <param><c>yor</c> is the new Point's y-coordinate.</param>
25     public Point(int xor, int yor) {
26         X = xor;
27         Y = yor;
28     }
29
30     /// <summary>This method changes the point's location to
31     /// the given coordinates.</summary>
32     /// <param><c>xor</c> is the new x-coordinate.</param>
33     /// <param><c>yor</c> is the new y-coordinate.</param>
34     /// <see cref="Translate"/>
35     public void Move(int xor, int yor) {
36         X = xor;
37         Y = yor;
38     }
39
40     /// <summary>This method changes the point's location by
41     /// the given x- and y-offsets.
42     /// <example>For example:
43     /// <code>
44     ///     Point p = new Point(3,5);
45     ///     p.Translate(-1,3);
46     /// </code>
47     /// results in <c>p</c>'s having the value (2,8).
48     /// </example>
49     /// </summary>
50     /// <param><c>xor</c> is the relative x-offset.</param>
51     /// <param><c>yor</c> is the relative y-offset.</param>
52     /// <see cref="Move"/>
53     public void Translate(int xor, int yor) {
54         X += xor;
55         Y += yor;
56     }
57
58     /// <summary>This method determines whether two Points have the same
59     /// location.</summary>
60     /// <param><c>o</c> is the object to be compared to the current
61     object.
62     /// </param>
63     /// <returns>True if the Points have the same location and they have
64     /// the exact same type; otherwise, false.</returns>
65     /// <seealso cref="operator==" />
66     /// <seealso cref="operator!=" />
67     public override bool Equals(object o) {
68         if (o == null) {
69             return false;
70         }
71     }

```

C# LANGUAGE SPECIFICATION

```
1         if (this == o) {
2             return true;
3         }
4         if (GetType() == o.GetType()) {
5             Point p = (Point)o;
6             return (X == p.X) && (Y == p.Y);
7         }
8         return false;
9     }
10    /// <summary>Report a point's location as a string.</summary>
11    /// <returns>A string representing a point's location, in the form
12    (x,y),
13    /// without any leading, training, or embedded whitespace.</returns>
14    public override string ToString() {
15        return "(" + X + ", " + Y + ")";
16    }
17    /// <summary>This operator determines whether two Points have the same
18    /// location.</summary>
19    /// <param><c>p1</c> is the first Point to be compared.</param>
20    /// <param><c>p2</c> is the second Point to be compared.</param>
21    /// <returns>True if the Points have the same location and they have
22    /// the exact same type; otherwise, false.</returns>
23    /// <seealso cref="Equals"/>
24    /// <seealso cref="operator!=">
25    public static bool operator==(Point p1, Point p2) {
26        if ((object)p1 == null || (object)p2 == null) {
27            return false;
28        }
29
30        if (p1.GetType() == p2.GetType()) {
31            return (p1.X == p2.X) && (p1.Y == p2.Y);
32        }
33
34        return false;
35    }
36    /// <summary>This operator determines whether two Points have the same
37    /// location.</summary>
38    /// <param><c>p1</c> is the first Point to be compared.</param>
39    /// <param><c>p2</c> is the second Point to be compared.</param>
40    /// <returns>True if the Points do not have the same location and the
41    /// exact same type; otherwise, false.</returns>
42    /// <seealso cref="Equals"/>
43    /// <seealso cref="operator==">
44    public static bool operator!=(Point p1, Point p2) {
45        return !(p1 == p2);
46    }
47    /// <summary>This is the entry point of the Point class testing
48    /// program.
49    /// <para>This program tests each method and operator, and
50    /// is intended to be run after any non-trivial maintenance has
51    /// been performed on the Point class.</para></summary>
52    public static void Main() {
53        // class test code goes here
54    }
55 }
56 }
```

E.4.2 Resulting XML

Here is the output produced by one documentation generator when given the source code for class `Point`, shown above:

```

1      <?xml version="1.0"?>
2      <doc>
3          <assembly>
4              <name>Point</name>
5          </assembly>
6          <members>
7              <member name="T:Graphics.Point">
8                  <remarks>Class <c>Point</c> models a point in a two-
9 dimensional
10                plane.
11                </remarks>
12            </member>
13            <member name="F:Graphics.Point.x">
14                <summary>Instance variable <c>x</c> represents the point's
15                x-coordinate.</summary>
16            </member>
17            <member name="F:Graphics.Point.y">
18                <summary>Instance variable <c>y</c> represents the point's
19                y-coordinate.</summary>
20            </member>
21            <member name="M:Graphics.Point.#ctor">
22                <summary>This constructor initializes the new Point to
23                (0,0).</summary>
24            </member>
25            <member name="M:Graphics.Point.#ctor(System.Int32,System.Int32)">
26                <summary>This constructor initializes the new Point to
27                (<paramref name="xor"/>,<paramref name="yor"/>).</summary>
28                <param><c>xor</c> is the new Point's x-coordinate.</param>
29                <param><c>yor</c> is the new Point's y-coordinate.</param>
30            </member>
31            <member name="M:Graphics.Point.Move(System.Int32,System.Int32)">
32                <summary>This method changes the point's location to
33                the given coordinates.</summary>
34                <param><c>xor</c> is the new x-coordinate.</param>
35                <param><c>yor</c> is the new y-coordinate.</param>
36                <see
37 cref="M:Graphics.Point.Translate(System.Int32,System.Int32)"/>
38            </member>
39            <member
40                name="M:Graphics.Point.Translate(System.Int32,System.Int32)">
41                <summary>This method changes the point's location by
42                the given x- and y-offsets.
43                <example>For example:
44                <code>
45                Point p = new Point(3,5);
46                p.Translate(-1,3);
47                </code>
48                results in <c>p</c>'s having the value (2,8).
49                </example>
50                </summary>
51                <param><c>xor</c> is the relative x-offset.</param>
52                <param><c>yor</c> is the relative y-offset.</param>
53                <see
54 cref="M:Graphics.Point.Move(System.Int32,System.Int32)"/>
55            </member>

```

C# LANGUAGE SPECIFICATION

```
1         <member name="M:Graphics.Point.Equals(System.Object)">
2             <summary>This method determines whether two Points have the
3 same
4             location.</summary>
5             <param><c>o</c> is the object to be compared to the current
6             object.
7             </param>
8             <returns>True if the Points have the same location and they
9 have
10            the exact same type; otherwise, false.</returns>
11            <seealso
12
13 cref="M:Graphics.Point.op_Equality(Graphics.Point,Graphics.Point)"/>
14            <seealso
15
16 cref="M:Graphics.Point.op_Inequality(Graphics.Point,Graphics.Point)"/>
17        </member>
18        <member name="M:Graphics.Point.ToString">
19            <summary>Report a point's location as a string.</summary>
20            <returns>A string representing a point's location, in the
21 form
22            (x,y),
23            without any leading, training, or embedded
24 whitespace.</returns>
25        </member>
26        <member
27
28 name="M:Graphics.Point.op_Equality(Graphics.Point,Graphics.Point)">
29            <summary>This operator determines whether two Points have the
30            same
31            location.</summary>
32            <param><c>p1</c> is the first Point to be compared.</param>
33            <param><c>p2</c> is the second Point to be compared.</param>
34            <returns>True if the Points have the same location and they
35 have
36            the exact same type; otherwise, false.</returns>
37            <seealso cref="M:Graphics.Point.Equals(System.Object)"/>
38            <seealso
39
40 cref="M:Graphics.Point.op_Inequality(Graphics.Point,Graphics.Point)"/>
41        </member>
42        <member
43
44 name="M:Graphics.Point.op_Inequality(Graphics.Point,Graphics.Point)">
45            <summary>This operator determines whether two Points have the
46            same
47            location.</summary>
48            <param><c>p1</c> is the first Point to be compared.</param>
49            <param><c>p2</c> is the second Point to be compared.</param>
50            <returns>True if the Points do not have the same location and
51            the
52            exact same type; otherwise, false.</returns>
53            <seealso cref="M:Graphics.Point.Equals(System.Object)"/>
54            <seealso
55
56 cref="M:Graphics.Point.op_Equality(Graphics.Point,Graphics.Point)"/>
57        </member>
58        <member name="M:Graphics.Point.Main">
59            <summary>This is the entry point of the Point class testing
60            program.
61            <para>This program tests each method and operator, and
62            is intended to be run after any non-trivial maintenance has
63            been performed on the Point class.</para></summary>
64        </member>
```

```
1      <member name="P:Graphics.Point.X">
2          <value>Property <c>X</c> represents the point's
3          x-coordinate.</value>
4      </member>
5      <member name="P:Graphics.Point.Y">
6          <value>Property <c>Y</c> represents the point's
7          y-coordinate.</value>
8      </member>
9  </members>
10 </doc>
```


F. Index

1

2 **This clause is informative.**

3 3 –

4 binary **168**

5 unary **161**

6 – –

7 pointer and **341**

8 postfix **152**

9 prefix **162**

10 ! **161**

11 != **171**

12 # *See* pre-processing directive, format of

13 % **165**

14 %= **182**

15 &

16 binary **176**

17 unary **340**

18 && **177**

19 & versus **177**

20 &= **182**

21 ()

22 cast operator *See* cast

23 grouping parentheses . *See* precedence:grouping

24 parentheses and

25 method call operator **148**

26 *

27 binary **163**

28 unary **338**

29 *= **182**

30 . **146**

31 / **164**

32 /* */ *See* comment, delimited

33 // *See* comment, single-line

34 /// *See* documentation comment

35 /= **182**

36 :

37 base class **218**

38 base interface **291**

39 ?: 179

40 @ identifier prefix **60**

41 @ verbatim string prefix **64**

42 []

43 array element access **150**

44 base access **152**

45 element access **149**

46 indexer access **150**

47 overloading element access *See* indexer

48 pointer element access **339**

49 \u *See* escape sequence, Unicode

50 \U *See* escape sequence, Unicode

51 ^ **176**

52 ^= **182**

53 | **176**

54 || **177**

55 | versus **177**

56 |= **182**

57 ~ **161**

58 +

59 binary **166**

60 unary **160**

61 ++

62 pointer and **341**

63 postfix **152**

64 prefix **162**

65 += **182**

66 event handler addition 38, **182**, 258

1	<	171	38	struct member.....	80
2	<<.....	170	39	accessibility domain.....	80
3	<<=.....	182	40	accessor.....	129
4	<=.....	171	41	event.....	260
5	=	179	42	add..260. <i>See also</i> +=, event handler addition	
6	-=.....	182	43	remove.....260. <i>See also</i> -=, event handler	
7	event handler removal.....	38, 182, 258	44	removal	
8	==.....	171	45	indexer.....	263, 321
9	>	171	46	get.....	263, 294
10	->.....	338	47	set.....	263, 294
11	>=.....	171	48	interface.....	294
12	>>.....	170	49	property.....	250, 251
13	arithmetic.....	170	50	get.....	251
14	logical.....	170	51	side-effects in a.....	255
15	>>=.....	182	52	set.....	251
16	0x integer literal prefix.....	62	53	address.....	334
17	OX integer literal prefix.....	62	54	address-of operator.....	<i>See</i> &, unary
18	abstract		55	analysis	
19	class and.....	217	56	lexical.....	56
20	event and.....	257	57	static flow.....	<i>See</i> static flow analysis
21	indexer and.....	263	58	application.....	7
22	method and.....	247	59	application domain.....	7
23	property and.....	251	60	application entry point.....	75
24	accessibility.....	32, 79	61	application parameter.....	75
25	class member.....	80	62	application startup.....	75
26	compilation unit type.....	80	63	application termination.....	75
27	constraints on.....	83	64	destructors and.....	76
28	enumeration member.....	80	65	exit status code.....	76
29	interface member.....	80	66	static variable and.....	105
30	internal.....	<i>See</i> internal	67	argument.....	7
31	Main 's.....	75	68	argument list.....	138
32	namespace type.....	80	69	ArrayTypeMismatchException and.....	140
33	private.....	<i>See</i> private	70	expression evaluation order.....	139
34	protected.....	<i>See</i> protected	71	method call.....	148
35	protected internal.....	<i>See</i> protected internal	72	overload resolution and.....	140
36	public.....	<i>See</i> public	73	variable length.....	<i>See</i> parameter array
37	restrictions on.....	80	74	ArithmeticException.....	318
			75	array.....	20, 287. <i>See also</i> Array

- 1 array of..... 20, **155**
- 2 base type of an **288**
- 3 creation of an *See new, array creation*
- 4 dimension of an 21, **154**
- 5 length of a **287**
- 6 element **106**
- 7 definite assignment and **106**
- 8 life of an..... **106**
- 9 type of an **287**
- 10 element access in an **150**
- 11 initializer for an 21, **289**
- 12 jagged *See array, array of*
- 13 multi-dimensional..... 20
- 14 rank of an..... 20, **154, 287**
- 15 rectangular 21
- 16 single-dimensional..... 20
- 17 subscript
- 18 types permitted in an..... **150**
- 19 subscripting.... *See also array, element access in*
- 20 an
- 21 Array..... 79, 123, 197, 288
- 22 conversion to **120**
- 23 members of **79, 288**
- 24 array covariance..... 139, **288**
- 25 array element **287**
- 26 `ArrayTypeMismatchException`
- 27 argument list and 139, 140
- 28 array covariance and..... 289
- 29 simple assignment and..... 180
- 30 as 171, **176**
- 31 cast versus..... **176**
- 32 assembly 7, 48, 209
- 33 assignment
- 34 compound 180, **182**
- 35 overloading **132**
- 36 definite **108**
- 37 try and..... **113**
- 38 when required **108**
- 39 event..... 180
- 40 simple..... 108, 179, **180**
- 41 associativity..... 28, **131**
- 42 grouping parentheses and..... 28, **131**
- 43 atomicity 118
- 44 attribute 52, **319**. *See also Attribute*
- 45 class naming convention **319**
- 46 compilation of an 325
- 47 delegate 323
- 48 event..... 323
- 49 add accessor 323
- 50 remove accessor 323
- 51 instance of an 325
- 52 method..... 323
- 53 name of an..... 322
- 54 property
- 55 get accessor 323
- 56 set accessor..... 323
- 57 reserved 326
- 58 specification of an 321
- 59 Attribute 319
- 60 attribute class 319
- 61 multi-use 319, 320
- 62 parameter
- 63 named 320
- 64 positional..... 320
- 65 single-use 319
- 66 attribute section..... **321**
- 67 Attribute suffix..... **323**
- 68 attribute target 323
- 69 assembly..... 323
- 70 event..... 323
- 71 field 323
- 72 method..... 323
- 73 param..... 323
- 74 property 323
- 75 return 323

1	type	323	39	boxing	22, 102
2	AttributeUsage.....	<i>See</i> AttributeUsageAttribute	40	break.....	31, 200
3	AttributeUsageAttribute	319, 326	41	do/while and.....	196
4	banker's rounding.....	100	42	finally and	200
5	base.....	151	43	for and	196
6	. and	151	44	inside nested iteration statements.....	200
7	[] and.....	151	45	reachability and	200
8	access member of	151	46	target of	200
9	constructor call		47	while and.....	195
10	explicit	270	48	byte.....	18, 97, 98 . <i>See also</i> Byte
11	implicit.....	270	49	Byte	78, 97
12	base class	<i>See</i> class, base	50	members of.....	78
13	base interface	<i>See</i> interface, base	51	<C>	463
14	behavior	7	52	C standard	5
15	implementation-defined.....	7	53	C++ standard.....	5
16	documenting	4	54	case	192
17	summary of all.....	377	55	goto	<i>See</i> goto case
18	undefined	3, 7	56	case label	192
19	unspecified.....	7	57	null as a	194
20	summary of all.....	378	58	cast	19, 121, 162
21	#define	69	59	as versus	176
22	block	30, 187	60	redundant.....	121
23	catch.....	203	61	catch	203
24	declaration in a	187	62	general.....	204
25	declaration space of a	<i>See</i> declaration space,	63	char.....	18, 97, 98 . <i>See also</i> Char
26	block and		64	integer literal and	98
27	empty	187	65	Char.....	79, 97
28	exiting a	199	66	members of.....	79
29	finally.....	199, 200, 201, 202, 203	67	character	
30	exception thrown from.....	205	68	carriage return	56
31	nested		69	encoding of.....	58
32	duplicate labels in a	77	70	form feed	58
33	duplicate local variables in a	76, 85	71	horizontal tab.....	58
34	simple name in a.....	145	72	line feed.....	56
35	try.....	203	73	line separator	56
36	bool	18, 97, 100 . <i>See also</i> Boolean	74	null	346
37	Boolean.....	79, 97	75	paragraph separator	56
38	members of	79			

1	Unicode class Cf.....	60	38	declaration space of a.....	<i>See</i> declaration space,
2	Unicode class Ll	59	39	class and	
3	Unicode class Lm	59	40	initialization of a	234
4	Unicode class Lo	59	41	initialization of a	43
5	Unicode class Lt	59	42	interface implementations and a.....	219
6	Unicode class Lu	59	43	member	79
7	Unicode class Mc	59	44	accessibility of a.....	216, 222
8	Unicode class Mn	59	45	constant	<i>See</i> constant
9	Unicode class Nl	59	46	constructor	
10	Unicode class Nd.....	59	47	instance	<i>See</i> constructor:instance
11	Unicode class Pc.....	59	48	static	<i>See</i> constructor:static
12	Unicode class Zs.....	58	49	destructor.....	<i>See</i> destructor
13	Unicode escape sequence	58	50	event.....	<i>See</i> event
14	vertical tab	58	51	field	<i>See</i> field
15	checked	98	52	hiding a.....	221
16	constant expression and.....	183	53	indexer.....	<i>See</i> indexer
17	explicit numeric conversion and.....	122	54	instance	222
18	integer addition and	166	55	method.....	<i>See</i> method
19	integer division and	164	56	operator	<i>See</i> operator
20	integer multiplication and.....	163	57	property	<i>See</i> property
21	integer subtraction and.....	168	58	static	222
22	operator.....	158	59	type.....	<i>See</i> type
23	shift operations and.....	170	60	members	220
24	statement.....	32, 206	61	nested	217
25	checked operator versus.....	206	62	non-abstract.....	217
26	unary minus and	161	63	permitted modifiers on a	217
27	class	17, 32, 101, 217	64	sealed.....	218, 219
28	abstract.....	44, 217	65	struct versus	279, 280
29	attribute.....	<i>See</i> attribute class	66	assignment.....	281
30	base	101, 218	67	boxing and unboxing.....	282
31	direct.....	218	68	constructors	282
32	accessibility of a	219	69	default values	281
33	classes which cannot be a	219	70	destructors	283
34	type accessibility	83	71	field initializers	282
35	circular dependence	219	72	inheritance.....	281
36	Console	15	73	meaning of this.....	282
37	declaration of.....	76	74	value semantics	280
			75	class library	7

1	CLI.....	iii, 15	39	versioning of a.....	232
2	CLS.....	11	40	constant expression	
3	<code>	463	41	default integral overflow checking.....	158
4	collection	197	42	constant folding.....	97
5	enumerating elements in a	<i>See</i> foreach	43	constructor	
6	System.Array	197	44	execution	
7	comment	56, 58	45	semantics of.....	272
8	delimited	57	46	instance	41, 269
9	documentation	<i>See</i> documentation comment	47	accessibility of a.....	269
10	single-line	57	48	default	273
11	Common Language Infrastructure.....	<i>See</i> CLI	49	initializer and.....	270
12	Common Language Specification	<i>See</i> CLS	50	private	273
13	compilation unit.....	209	51	invocation of a.....	138
14	attributes of a	209	52	overloading of a	84
15	interdependency of	209	53	parameterless	
16	type accessibility and.....	216	54	struct and	282
17	Conditional	<i>See</i> ConditionalAttribute	55	signature of a.....	84
18	conditional compilation	69 . <i>See also</i>	56	static	43, 274
19	ConditionalAttribute		57	struct	
20	conditional compilation symbol	67	58	this in a	283
21	defining a	<i>See</i> #define	59	value type	96
22	scope of a.....	67	60	continue.....	31, 200
23	undefining a	<i>See</i> #undefine	61	do/while and.....	196
24	ConditionalAttribute.....	326	62	finally and	200
25	conformance	3	63	for and	196
26	Console.....	15	64	inside nested iteration statements.....	200
27	const.....	97. <i>See also</i> constant	65	reachability and	200
28	constant.....	34, 228	66	target of	200
29	accessibility of a	228	67	while and	195
30	initializer for a	228	68	conversion	19, 119
31	interdependency of	229	69	better	141
32	local	30	70	boxing	102, 120
33	declaration of	189	71	explicit.....	19, 121
34	scope of.....	190, 194	72	enumeration.....	122
35	named	<i>See</i> enum	73	numeric	121
36	readonly versus.....	229, 231	74	reference.....	123
37	restrictions on type of a	228	75	standard	124
38	type accessibility of a	83			

- 1 user-defined **124, 126**
- 2 using a cast **162**
- 3 using as **176**
- 4 identity **119**
- 5 implicit 19, **119**
- 6 constant expression **120**
- 7 constant int to byte **120**
- 8 constant int to sbyte **120**
- 9 constant int to short **120**
- 10 constant int to ushort **120**
- 11 decimal to/from floating-point **100**
- 12 enumeration **120**
- 13 numeric **119**
- 14 pre-defined and exceptions **119**
- 15 reference **120**
- 16 standard **124**
- 17 to/from char **98, 120**
- 18 user-defined 119, **121, 125**
- 19 zero to enumeration **120**
- 20 standard **124**
- 21 to/from bool **100**
- 22 unboxing **103, 123**
- 23 incompatible type and **103**
- 24 null and **103**
- 25 user-defined **124, 268**
- 26 evaluation of a **124**
- 27 worse **142**
- 28 conversion operator *See* operator, conversion
- 29 creation of an instance *See* new, object creation
- 30 cref **462**
- 31 Current 197
- 32 d real literal suffix **63**
- 33 D real literal suffix **63**
- 34 decimal 18, 97, **99**. *See also* Decimal
- 35 Decimal 79, 97
- 36 members of **79**
- 37 declaration **76**
- 38 name hiding by a **76**
- 39 order of **77**
- 40 type *See* type, declaration of a
- 41 declaration space **76**
- 42 block and **76, 77**
- 43 class and **76**
- 44 duplicate names in a **76**
- 45 enumeration and **76**
- 46 global **76**
- 47 interface and **76**
- 48 label **77, 188**
- 49 local variable **76**
- 50 namespace **76, 77**
- 51 nested blocks and **77**
- 52 struct and **76**
- 53 switch block and **76, 77**
- 54 *default* **192**
- 55 goto *See* goto default
- 56 default label **192**
- 57 #define **68**
- 58 ConditionalAttribute and **326**
- 59 definite assignment *See* assignment, definite
- 60 definitely assigned *See* variable, definitely
- 61 assigned
- 62 definitions **7**
- 63 delegate 17, 38, 47, **311**. *See also* Delegate. *See*
- 64 *also* Delegate
- 65 accessibility of a **311**
- 66 combination of a **168**
- 67 creation of a *See* new, delegate creation
- 68 equality of *See* operator, equality, delegate
- 69 invocation of a **149, 313**
- 70 removal of a **169**
- 71 sealedness of a **312**
- 72 Delegate 47, 79, 123, 311
- 73 conversion to **120**
- 74 members of **79**
- 75 derived class **219**. *See* class, derived

1	design goals	iii	38	ID string	469
2	destructor	42, 276	39	processing of	469
3	instance variable and	105	40	documentation generator.....	461
4	invocation of a	276	41	documentation viewer.....	461
5	struct and	283	42	double.....	18, 97, 98 . <i>See also</i> Double
6	diagnostic message	7	43	precision.....	99
7	Dispose	207	44	range.....	99
8	DivideByZeroException.....	317	45	Double.....	79, 97
9	decimal division.....	165	46	members of.....	79
10	decimal remainder and.....	166	47	element access.....	149
11	integer division and	164	48	#elif	69
12	integer remainder and.....	165	49	#else	69
13	do/while	31, 195	50	else	<i>See</i> if/else
14	break and	196	51	#endif	69
15	continue and.....	196	52	#endregion.....	72
16	reachability and	196	53	enum.....	17, 47, 307 . <i>See also</i> Enum
17	documentation comment	461	54	accessibility and	307
18	recommended tags in.....	462	55	declaration of an.....	76
19	XML output from	476	56	declaration space of an.....	<i>See</i> declaration space,
20	documentation comment tag		57	enumeration	
21	<c>	<i>See</i> <c>	58	member	308
22	<code>	<i>See</i> <code>	59	initialization of an	308
23	<example>	<i>See</i> <example>	60	value of an.....	308
24	<exception>	<i>See</i> <exception>	61	members of an.....	79
25	<list>.....	<i>See</i> <list>	62	permitted operations on an.....	310
26	<para>.....	<i>See</i> <para>	63	underlying type of an	307
27	<param>.....	<i>See</i> <param>	64	value of an.....	310
28	<paramref>	<i>See</i> <paramref>	65	Enum.....	310
29	<permission>	<i>See</i> <permission>	66	#error.....	3, 71
30	<remarks>.....	<i>See</i> <remarks>	67	error	
31	<returns>	<i>See</i> <returns>	68	compile-time	7
32	<see>	<i>See</i> <see>	69	escape sequence	
33	<seealso>	<i>See</i> <seealso>	70	alert	64
34	<summary>.....	<i>See</i> <summary>	71	backslash	64
35	<value>	<i>See</i> <value>	72	backspace	64
36	cref.....	<i>See</i> cref	73	carriage return	64
37	documentation file	461	74	double quote.....	64
			75	form feed	64

1	hexadecimal.....	63	39	types thrown by certain C# operations.....	318
2	regular string literal and.....	64	40	Exception	317
3	verbatim string literal and.....	64	41	catch and	203
4	horizontal tab	64	42	throw and	202
5	list of.....	64	43	Exception.Exception	317
6	new line	64	44	Exception.InnerException.....	317
7	null.....	64	45	Exception.Message	317
8	simple	63	46	Execution Order	93
9	regular string literal and.....	64	47	expanded form ..See function member, applicable,	
10	verbatim string literal and.....	64	48	expanded form	
11	single quote.....	64	49	explicit.....	268
12	Unicode.....	58, 59	50	expression	129
13	vertical tab	64	51	array creation.....See new, array creation	
14	event	38, 257	52	boolean	184
15	abstract.....	262	53	constant	183
16	accessibility of a	257	54	delegate creation	See new, delegate creation
17	accessing an	137	55	event access.....	129
18	external	258	56	indexer access	129
19	handler	257	57	invocation.....	148
20	inhibiting overriding of an.....	262	58	kinds of.....	129
21	instance	261	59	method group	129
22	interface and	294	60	namespace	129
23	override.....	262	61	nothing	129
24	sealed	262	62	object creation..... See new, object creation	
25	static.....	261	63	parenthesized.....	146
26	type accessibility of an	84	64	primary	143
27	virtual.....	261	65	property access.....	129
28	event access expression	See expression, event	66	type.....	129
29	access		67	value.....	129
30	<example>	463	68	value of an.....	130
31	examples	13	69	variable.....	129
32	<exception>	463	70	extensions.....	3
33	exception.....	7, 317. See also Exception	71	documenting.....	4
34	catching from other languages.....	See catch,	72	extern.....	248
35	general		73	event and	257
36	handling of an	317	74	indexer and.....	262
37	propagation of an.....	202	75	property and	250, 251
38	rethrow an.....	See throw, with no expression	76	f real literal suffix.....	63

1	F real literal suffix	63	38	applicable	141
2	false	61, 100	39	expanded form.....	141
3	field.....	23, 35, 229	40	normal form.....	141
4	accessibility of a	230	41	better	141
5	initialization of a.....	233, 234	42	invocation of a.....	142
6	initializer for a	230	43	naming restrictions on a	220
7	instance	231	44	function pointer.....	<i>See</i> delegate
8	initialization of an.....	236	45	garbage collection	27
9	public		46	destructor call and	42
10	property versus	253	47	fixed variables and	336
11	readonly	35	48	movable variables and.....	336
12	versioning of a	232	49	pointer tracking and	334
13	static.....	231	50	garbage collector	91
14	initialization of a.....	234	51	get accessor	37
15	type accessibility of a	83	52	attribute property.....	323
16	volatile	232	53	indexer.....	263, 294
17	finalization		54	property	251
18	suppression of.....	76	55	GetEnumerator	197
19	finally		56	global name	<i>See</i> declaration space, global
20	break and	200	57	goto	30, 200
21	continue and.....	200	58	finally and	201
22	goto and	201	59	label and	188
23	jump statement and.....	199	60	reachability and	201
24	return and.....	202	61	target of	201
25	financial calculations	<i>See</i> decimal	62	goto case.....	194, 200
26	float.....	18, 97, 98. See also Single	63	goto default	194, 200
27	precision	99	64	grammar	55
28	range	99	65	lexical.....	9, 55
29	for	31, 196	66	syntactic	9, 55
30	break and	196	67	ICloneable	
31	continue and.....	196	68	conversion to.....	120
32	reachability and	197	69	identifier.....	58, 59
33	for condition	196	70	beginning with two underscores.....	60
34	for initializer	196	71	verbatim	60
35	for iterator	196	72	IDisposable	207
36	foreach	31, 197	73	IEC	<i>See</i> International Electrotechnical
37	function member.....	136, 220	74	Commission	
			75	IEC 60559 standard.....	5

- 1 IEEE *See* Institute of Electrical and Electronics
2 Engineers
- 3 IEEE 754 standard *See* IEC 60559 standard
- 4 IEnumerable.GetEnumerator.. *See* GetEnumerator
- 5 IEnumerator.Current.....*See* Current
- 6 IEnumerator.MoveNext.....*See* MoveNext
- 7 #if..... **69**
- 8 if/else 30, **190**
- 9 reachability and **191**
- 10 implementation **8**
- 11 conforming **3**
- 12 implicit..... **268**
- 13 indexer 40, **262**
- 14 accessibility of an **262**
- 15 accessing an 137
- 16 interface and **294**
- 17 output parameter and **138**
- 18 overloading of an **85**
- 19 property versus **263**
- 20 reference parameter and..... **138**
- 21 signature of an **84**
- 22 type accessibility of an **84**
- 23 indexer access **150**
- 24 indexer access expression. *See* expression, indexer
25 access
- 26 IndexOutOfRangeException
- 27 array access and 150
- 28 infinity
- 29 negative..... **99**
- 30 positive **99**
- 31 informative text **3**
- 32 inheritance 43, **101, 221**
- 33 initializer
- 34 array 154, **289**
- 35 constant
- 36 local **189**
- 37 constructor
- 38 instance **270**
- 39 static **274**
- 40 enum member..... **308**
- 41 field **230**
- 42 fixed pointer **343**
- 43 for **196**
- 44 stack allocation..... **346**
- 45 struct..... **282**
- 46 variable
- 47 instance **271**
- 48 local..... **189**
- 49 static **274**
- 50 initially assigned ... *See* variable, initially assigned
- 51 initially unassigned *See* variable, initially
52 unassigned
- 53 instance **101**
- 54 absence of..... *See* null
- 55 Institute of Electrical and Electronics Engineers 11
- 56 int 18, 97, **98**. *See also* Int32
- 57 Int16 79, 97
- 58 members of..... **79**
- 59 Int32 19, 79, 97
- 60 members of..... **79**
- 61 Int64 79, 97
- 62 members of..... **79**
- 63 interface..... 17, 45, **291**
- 64 abstract class and..... **304**
- 65 accessibility of an..... **291**
- 66 base **291**
- 67 type accessibility **83**
- 68 declaration of..... **76**
- 69 declaration space of a..... *See* declaration space,
70 interface
- 71 implementation of an **296**
- 72 inheritance and **302**
- 73 mapping to an..... **299**
- 74 member **79, 292**
- 75 accessibility of an..... **292**
- 76 event **294**

1	indexer	294	39	string	64
2	method	293	40	duplicate memory sharing	65
3	property.....	293	41	regular	64
4	name of an	296	42	verbatim	64
5	re-implementation of an.....	303	43	lock.....	32, 206
6	internal.....	33, 80, 81	44	long	18, 97, 98. See also Int64
7	International Electrotechnical Commission.....	11	45	lu integer literal suffix.....	62
8	International Organization for Standardization .	11	46	IU integer literal suffix.....	62
9	InvalidCastException		47	Lu integer literal suffix	62
10	explicit reference conversion.....	123	48	LU integer literal suffix.....	62
11	unboxing and	103	49	lvalue.....	<i>See variable reference</i>
12	is 102, 129, 171, 175		50	m real literal suffix	63
13	ISO.....	<i>See International Organization for</i>	51	M real literal suffix	63
14	Standardization		52	Main	15, 75
15	ISO/IEC 10646	3, 5	53	accessibility of.....	75
16	keyword	58, 60	54	command-line arguments and	75
17	use as an identifier	60	55	optional parameter in	75
18	l integer literal suffix	62	56	overloading of	75
19	L integer literal suffix.....	62	57	recognized signatures for	75
20	label	30, 188	58	return type int.....	76
21	declaration of a	77	59	return type void.....	76
22	goto and	200	60	selecting from multiple	75
23	scope of a.....	188	61	member	78
24	library	<i>See class library</i>	62	nested	80
25	#line	72	63	overloading of a	<i>See overloading</i>
26	line terminator	56	64	scope of a	<i>See scope</i>
27	<list>.....	464	65	top-level	80
28	literal.....	18, 58, 61, 144	66	unsafe	332
29	boolean	61	67	member access	146. See accessibility
30	character	63	68	member lookup	135
31	decimal	<i>See literal:real</i>	69	member name	
32	floating-point	<i>See literal:real</i>	70	form of a.....	78
33	integer	61	71	forward reference	86
34	decimal	61	72	memory management.....	91
35	hexadecimal.....	61	73	automatic.....	26
36	type of an	62	74	direct	26, 347
37	null.....	65	75	method.....	36, 236
38	real	62			

- 1 abstract..... **247**
- 2 accessibility of a **237**
- 3 calling a **148**
- 4 conditional**326**. *See* ConditionalAttribute
- 5 external **248**
- 6 inhibiting overriding of a.....*See* sealed, method
- 7 instance 36, **243**
- 8 invocation of a 136, **148, 238**
- 9 non-void..... **249**
- 10 overloading of a 36, **84**
- 11 overridden base..... **245**
- 12 override..... **245**
- 13 overriding *See* virtual
- 14 sealed 246
- 15 signature of a 36, **84**
- 16 static..... 36, **243**
- 17 virtual..... **243**
- 18 void **237, 249**
- 19 return and..... **202**
- 20 method group expression..*See* expression, method
- 21 group
- 22 modifier
- 23 abstract..... *See* abstract
- 24 default *See* modifier, none
- 25 extern*See* extern
- 26 internal.....*See* internal
- 27 new *See* new
- 28 none **80**
- 29 out.....*See* out
- 30 override..... *See* override
- 31 params.....*See* parameter array
- 32 private*See* private
- 33 protected *See* protected
- 34 protected internal*See* protected internal
- 35 public*See* public
- 36 readonly*See* readonly
- 37 ref..... *See* ref
- 38 sealed*See* sealed
- 39 static *See* static
- 40 virtual*See* virtual
- 41 volatile..... *See* volatile
- 42 monetary calculations*See* decimal
- 43 MoveNext 197
- 44 mutex..... *See* lock
- 45 mutual exclusion lock *See* lock
- 46 name
- 47 hiding **85, 87**
- 48 via inheritance **88**
- 49 via nesting **87**
- 50 qualified **89**
- 51 fully **90**
- 52 simple..... **144**
- 53 visibility of a **87**
- 54 namespace **8, 15, 48, 76, 80, 209**
- 55 accessibility *See* accessibility, namespace
- 56 alias for a..... **211**
- 57 declaration of a..... 209
- 58 global..... 209, 210
- 59 import members from a..... **213**
- 60 members of a..... **215**
- 61 modifiers and..... 210
- 62 name
- 63 form of a..... 210
- 64 nested 210
- 65 type
- 66 accessibility and **216**
- 67 namespace expression .*See* expression, namespace
- 68 NaN 99
- 69 nested member *See* member, nested
- 70 nested scope *See* scope, nested
- 71 new **153**
- 72 array creation..... 154
- 73 class member hiding and..... **221**
- 74 delegate creation 155, 313
- 75 dimension length evaluation order **154**

1	object creation	153	39	operator	28, 39, 40, 58, 66 , 130 , 131, 265
2	value type and.....	96	40	binary	<i>See</i> -, binary
3	new, array creation	21	41	unary	<i>See</i> -, unary
4	normal form	<i>See</i> function member, applicable,	42	--	
5	normal form		43	postfix	<i>See</i> --, postfix
6	normative text.....	3 , 13	44	prefix	<i>See</i> --, prefix
7	conditionally	3 , 13	45	! <i>See</i> !	
8	Not-a-Number.....	<i>See</i> NaN	46	!= <i>See</i> !=	
9	notes.....	13	47	% <i>See</i> %	
10	nothing expression.....	<i>See</i> expression, nothing	48	%=	<i>See</i> %=
11	null.....	65	49	&	
12	representation of	108	50	binary	<i>See</i> &:binary
13	NullReferenceException		51	unary	<i>See</i> &, unary
14	array access and.....	150	52	&&	<i>See</i> &&
15	delegate creation and	156	53	&=	<i>See</i> &=
16	delegate invocation and	149	54	()	
17	foreach and	197	55	cast	<i>See</i> (), cast operator
18	function member		56	method call.....	<i>See</i> (), method call operator
19	invocation and	143	57	*	
20	member access and.....	147	58	binary	<i>See</i> *, binary
21	throw null and.....	202	59	unary	<i>See</i> *, unary
22	object	17, 22, 79, 95 , 101. <i>See also</i> Object	60	*=	<i>See</i> *=
23	aliasing of	95	61	. <i>See</i> .	
24	as a direct base class.....	219	62	/ <i>See</i> /	
25	conversion of value type to.....	120	63	/= <i>See</i> /=	
26	conversion to	120	64	?: <i>See</i> ?:	
27	conversion to value type.....	123	65	[] <i>See</i> []	
28	inaccessible.....	91	66	pointer element access	<i>See</i> [], pointer element
29	live	91	67	access	
30	Object	79, 101	68	^ <i>See</i> ^	
31	members of	79	69	^=	<i>See</i> ^=
32	object creation	<i>See</i> new, object creation	70	<i>See</i>	
33	Obsolete.....	<i>See</i> ObsoleteAttribute	71	<i>See</i>	
34	ObsoleteAttribute	329	72	= <i>See</i> =	
35	operand	130	73	~ <i>See</i> ~	
36	mixing decimal and floating-point	100	74	+	
37	mixing integral and decimal	100	75	binary	<i>See</i> +, binary
38	mixing integral and floating-point.....	99	76	unary	<i>See</i> +, unary

1	++	38	equality
2	postfix..... <i>See</i> ++, postfix	39	boolean..... 173
3	prefix..... <i>See</i> ++, prefix	40	delegate..... 175
4	+ =..... <i>See</i> +=	41	reference..... 173
5	< <i>See</i> <	42	string..... 174
6	<<..... <i>See</i> <<	43	external..... 266
7	<<=..... <i>See</i> <<=	44	floating-point
8	<=..... <i>See</i> <=	45	exceptions and..... 99
9	= <i>See</i> =	46	hiding of an..... 88
10	= = <i>See</i> ==	47	integral overflow checking and..... 158
11	==..... <i>See</i> ==	48	invocation of an..... 137
12	> <i>See</i> >	49	is 102. <i>See</i> is
13	->..... <i>See</i> ->	50	logical..... 176
14	>=..... <i>See</i> >=	51	boolean..... 177
15	>>..... <i>See</i> >>	52	conditional..... 177
16	>>=..... <i>See</i> >>=	53	boolean..... 178
17	address-of..... <i>See</i> &, unary	54	user-defined..... 178
18	arithmetic..... 163	55	enumeration..... 177
19	as <i>See</i> as	56	integer..... 176
20	assignment..... <i>See</i> assignment	57	new..... <i>See</i> new
21	associativity of an..... <i>See</i> associativity	58	order of evaluation of..... 130
22	binary..... 130	59	overloading an..... 19
23	integral types and..... 98	60	restrictions on..... 132
24	overload resolution..... 133	61	overloading of an..... 85, 130, 131
25	overloadable..... 131	62	restrictions on..... 132
26	overloading..... 267	63	overloading an..... 265
27	bitwise complement..... <i>See</i> ~	64	precedence of..... <i>See</i> precedence
28	cast..... <i>See</i> cast	65	relational..... 171
29	overloading..... <i>See</i> conversion, user-defined	66	shift..... 170
30	checked..... <i>See</i> checked, operator	67	signature of an..... 84
31	comparison..... 171	68	sizeof..... <i>See</i> sizeof
32	decimal..... 172	69	ternary..... 130
33	enumeration..... 173	70	typeof..... <i>See</i> typeof
34	floating-point..... 172	71	unary..... 130
35	integer..... 171	72	integer types and..... 98
36	conditional..... <i>See</i> ?:	73	overload resolution..... 132
37	conversion..... 124, 268	74	overloadable..... 131

1	overloading	267	38	overload resolution.....	136, 140
2	unchecked.....	<i>See</i> unchecked, operator	39	overloading	84
3	user-defined	133	40	override	135, 143, 245
4	output parameter and	138	41	base access and.....	152
5	reference parameter and.....	138	42	<para>	465
6	order of declarations	<i>See</i> declaration, order of	43	<param>	465
7	order of evaluation		44	parameter.....	8, 237
8	argument list expressions.....	139	45	output	24, 106, 108, 138, 239
9	operands in an expression.....	130	46	definite assignment and.....	106
10	operators	<i>See</i> operator, order of evaluation of	47	this as an.....	107
11	out.....	24, 106, 138, 239	48	reference.....	24, 106, 108, 138, 238
12	signature and.....	84	49	definite assignment and.....	106
13	OutOfMemoryException		50	this as a.....	106
14	array creation and	155	51	type accessibility of a.....	84
15	delegate creation and	156	52	value.....	23, 106, 138, 238
16	object creation and.....	154	53	definite assignment and.....	106
17	string concatenation and	167	54	life of a	106
18	output.....	15	55	parameter array	25, 139, 240
19	formatted.....	16	56	signature and	84
20	overflow.....	20	57	<paramref>.....	466
21	checking of integer	98, 158, 206	58	params	25, 240. <i>See</i> parameter array
22	pointer increment or decrement	341, 377	59	<permission>.....	466
23	OverflowException		60	pointer	
24	array creation and	155	61	address difference of	341
25	checked operator and	158, 159	62	arithmetic and.....	341
26	decimal addition and.....	167	63	comparison of.....	342
27	decimal and.....	100	64	decrementing a	341
28	decimal division.....	165	65	fixed	
29	decimal remainder and.....	166	66	initializer	343
30	decimal subtraction and	168	67	incrementing a.....	341
31	explicit numeric conversion and.....	122	68	indirection of a	338
32	integer addition and	166	69	member access via a.....	338
33	integer division and	164	70	permitted operations on a.....	335
34	integer subtraction and.....	168	71	referent type of a	334
35	integral types and.....	98	72	string	
36	multiplication and	163, 164	73	writing through a.....	346
37	unary minus and	161	74	to function	<i>See</i> delegate

- 1 to member function.....*See* delegate
- 2 type of a 333
- 3 precedence 28, **130**
- 4 grouping parentheses and **131**
- 5 precedence table **130**
- 6 pre-processing declaration..... **68**
- 7 permitted placement of **68**
- 8 pre-processing directive..... 56
- 9 #define *See* #define
- 10 #elif..... *See* #elif
- 11 #else.....*See* #else
- 12 #endif.....*See* #endif
- 13 #endregion*See* #endregion
- 14 #error *See* #error
- 15 #if..... *See* #if
- 16 #line*See* #line
- 17 #region.....*See* #region
- 18 #undef.....*See* #undef
- 19 #warning*See* #warning
- 20 conditional compilation **69**
- 21 nesting of **70**
- 22 ordering of in a set **70**
- 23 format of 66
- 24 pre-processing expression **67**
- 25 evaluation rules..... **68**
- 26 grouping parentheses in **67**
- 27 operators permitted in..... **67**
- 28 private 33, **80**, 81
- 29 production..... 9
- 30 program..... **8**, **55**, **209**
- 31 conforming **4**
- 32 strictly conforming **3**
- 33 valid **8**
- 34 program entry point 15
- 35 program instantiation..... **8**
- 36 programming language
- 37 interfacing with another..... 60
- 38 promotion
- 39 numeric **133**
- 40 binary **134**
- 41 unary **134**
- 42 property 37, **250**
- 43 abstract **255**
- 44 accessibility of a..... **250**
- 45 accessing a 136
- 46 external..... **251**
- 47 indexer versus **263**
- 48 inhibiting overriding of a **256**
- 49 inlining possibilities of..... **254**
- 50 instance **251**
- 51 interface and..... **293**
- 52 output parameter and..... **138**
- 53 override **256**
- 54 public field versus **253**
- 55 read-only **252**
- 56 read-write **252**
- 57 reference parameter and **138**
- 58 sealed..... **256**
- 59 static **251**
- 60 type accessibility of a..... **84**
- 61 virtual **255**
- 62 write-only **252**
- 63 property access expression..... *See* expression,
- 64 property access
- 65 protected..... 33, **80**, 81, 83
- 66 protected internal..... 33, **80**, 81, 83
- 67 public..... 33, **80**, 81
- 68 punctuator 58, **66**
- 69 reachability..... **185**
- 70 readonly..... 35, 97, **231**
- 71 constant versus 229, **231**
- 72 recommended practice **8**
- 73 ref 24, 106, 138, 238
- 74 signature and **84**
- 75 reference..... **95**

1	equality of..... <i>See</i> operator, equality, reference	38	method and.....	246
2	reference parameter <i>See</i> parameter, reference	39	property and	250
3	#region.....	72	string types and	101
4	region.....	72	value types and.....	96
5	<remarks>.....	466	<see>.....	467
6	reserved word <i>See</i> keyword	43	<seealso>.....	468
7	resource.....	207	set accessor.....	37
8	<i>disposal of a</i>	207	attribute property.....	323
9	return	32, 202	indexer.....	263, 294
10	finally and.....	202	property	251
11	from void Main.....	76	short.....	18, 97, 98. <i>See also</i> Int16
12	reachability and	202	side effect	93
13	with expression	202	signature	84
14	with no expression	202	Single	79, 97
15	return type		members of.....	79
16	type accessibility of a	83, 84	sizeof	130, 342
17	<returns>	467	source file.....	8, 55, 209
18	sbyte.....	18, 97, 98. <i>See also</i> SByte	declaration space and multiple.....	76
19	SByte	78, 97	line number in a.....	72
20	members of	78	name of a.....	72
21	scope.....	85	type suffix .cs	15
22	class member	85	59 source text	
23	enum member	86	60 exclusion of.....	69
24	inner.....	85	61 inclusion of.....	69
25	label	86	62 stackalloc.....	346
26	local variable	86	63 freeing memory obtained via	346
27	local variable in for.....	86	64 StackOverflowException	318
28	namespace member	85	65 stackalloc and.....	346
29	nested.....	85	66 statement	29, 185
30	parameter	86	67 break.....	<i>See</i> break
31	struct member	86	68 checked	<i>See</i> checked, statement
32	using name.....	85	69 composite	185
33	sealed		70 continue.....	<i>See</i> continue
34	abstract class and	218	71 declaration.....	188
35	class and.....	218	72 do/while.....	<i>See</i> do/while
36	event and.....	257	73 embedded	185
37	indexer and	262	74 empty.....	187

1	end point of.....	185	38	boxing and.....	282
2	reachability of.....	187	39	class versus.....	279, 280
3	expression.....	190	40	assignment.....	281
4	for.....	<i>See for</i>	41	boxing and unboxing.....	282
5	foreach.....	<i>See foreach</i>	42	constructors.....	282
6	goto.....	<i>See goto</i>	43	default values.....	281
7	if/else.....	<i>See if/else</i>	44	destructors.....	283
8	iteration.....	195	45	field initializers.....	282
9	jump.....	199	46	inheritance.....	281
10	target of a.....	199	47	meaning of this.....	282
11	try statement and.....	199	48	value semantics.....	280
12	labeled.....	188	49	declaration of.....	76
13	lock.....	<i>See lock</i>	50	declaration space of a.....	<i>See declaration space,</i>
14	reachable.....	185	51	struct	
15	return.....	<i>See return</i>	52	field alignment in a.....	343
16	selection.....	190	53	field initializers and.....	282
17	switch.....	<i>See switch</i>	54	inheritance and.....	281
18	throw.....	<i>See throw</i>	55	interfaces and.....	279
19	try.....	<i>See try</i>	56	member.....	78, 280
20	unchecked.....	<i>See unchecked, statement</i>	57	accessibility of a.....	216
21	unreachable.....	185	58	padding in a.....	343
22	unsafe.....	331	59	pass by reference.....	281
23	using.....	<i>See using-statement</i>	60	pass by value.....	281
24	while.....	<i>See while</i>	61	permitted modifiers on a.....	279
25	statement list.....	187	62	return by value.....	281
26	static.....	15, 105, 222	63	unboxing and.....	282
27	static flow analysis.....	108, 186	64	<summary>.....	468
28	string.....	17, 101. <i>See also String</i>	65	switch.....	31, 191
29	concatenation of.....	167	66	governing type of.....	192
30	C-style.....	346	67	string as.....	194
31	equality of.....	<i>See operator, quality, string</i>	68	reachability and.....	194
32	null-terminated.....	346	69	switch block.....	192
33	String.....	79. <i>See String</i>	70	declaration space of a.....	<i>See declaration space,</i>
34	members of.....	79	71	switch block	
35	struct.....	17, 44, 279	72	simple name in a.....	145
36	advice for using over class.....	279	73	switch label.....	192
37	assignment and.....	281	74	switch section.....	192
			75	end point of	

1	reachability of.....	187	40	System.OverflowException	<i>See</i>
2	symbol		41	OverflowException	
3	non-terminal	9	42	System.SByte	<i>See</i> SByte
4	terminal.....	9	43	System.Single.....	<i>See</i> Single
5	System	15, 97	44	System.StackOverflowException	<i>See</i>
6	System.ArithmeticException.....	<i>See</i>	45	StackOverflowException	
7	ArithmeticException		46	System.Type.....	<i>See</i> Type
8	System.Array	<i>See</i> Array	47	System.TypeInitializationException	<i>See</i>
9	System.ArrayTypeMismatchException.....	<i>See</i>	48	TypeInitializationException	
10	ArrayTypeMismatchException		49	System.UInt16	<i>See</i> UInt16
11	System.Attribute.....	326. <i>See</i> Attribute	50	System.UInt32	<i>See</i> UInt32
12	System.AttributeUsageAttribute.....	<i>See</i>	51	System.UInt64	<i>See</i> UInt64
13	AttributeUsageAttribute		52	this.....	106, 107, 151
14	System.Boolean	<i>See</i> Boolean	53	assignment to in struct.....	282
15	System.Byte.....	<i>See</i> Byte	54	constructor call	
16	System.Char.....	<i>See</i> Char	55	explicit.....	270
17	System.ConditionalAttribute	<i>See</i>	56	indexer and.....	263
18	ConditionalAttribute		57	this access.....	151
19	System.Console	<i>See</i> Console	58	throw	32, 202
20	System.Decimal.....	<i>See</i> Decimal	59	reachability and.....	202
21	System.Delegate	<i>See</i> Delegate. <i>See</i> Delegate	60	with expression	202
22	System.DivideByZeroException	<i>See</i>	61	with no expression	202
23	DivideByZeroException		62	throw point	202
24	System.Double.....	<i>See</i> Double	63	token.....	9 , 56, 58
25	System.Enum.....	<i>See</i> Enum	64	separation of.....	56
26	System.Exception	<i>See</i> Exception	65	top-level member	<i>See</i> member, top-level
27	System.IDisposable	<i>See</i> IDisposable	66	ToString	22, 36
28	System.IndexOutOfRangeException.....	<i>See</i>	67	string concatenation and.....	167
29	IndexOutOfRangeException		68	true	61 , 100
30	System.Int16.....	<i>See</i> Int16	69	try	203 , 317
31	System.Int32.....	<i>See</i> Int32	70	jump statement and	199
32	System.Int64.....	<i>See</i> Int64	71	reachability and.....	206
33	System.InvalidCastException.....	<i>See</i>	72	try block	<i>See</i> block, try
34	InvalidCastException		73	type.....	95
35	System.NullReferenceException.....	<i>See</i>	74	array	101 , 287
36	NullReferenceException		75	array element.....	287
37	System.ObsoleteAttribute... <i>See</i> ObsoleteAttribute		76	base	135
38	System.OutOfMemoryException	<i>See</i>	77	boolean.....	100
39	OutOfMemoryException				

- 1 versus integer types **100**
- 2 class *See* class
- 3 collection *See* collection
- 4 compile-time 243
- 5 constituent **222**
- 6 decimal **99**
- 7 precision **99**
- 8 range **99**
- 9 representation of **100**
- 10 versus floating-point **100**
- 11 declaration of a 76, **215**
- 12 delegate **102**
- 13 dynamic 102
- 14 check *See* is
- 15 element **197**
- 16 enum *See* enum
- 17 enumeration 95
- 18 enumeration 100
- 19 representation of 100
- 20 floating-point
- 21 versus decimal **100**
- 22 floating-point 18, **98**
- 23 representation of **98**
- 24 heap allocation and 279
- 25 initialization of
- 26 static variable and 105
- 27 integer 18, **97**
- 28 char differences **98**
- 29 representation of **97**
- 30 interface **101**
- 31 memory occupied by *See* sizeof
- 32 nested 215, **223**
- 33 non-nested **223**
- 34 null
- 35 conversion from **120**
- 36 object **101**
- 37 object as base class of every 95
- 38 pointer 95. *See* pointer, type
- 39 reference 16, 95, **100**
- 40 null compatibility with **101**
- 41 value versus 95
- 42 referent **334**
- 43 run-time 243
- 44 compatibility check *See* is
- 45 sealed 96
- 46 simple **95, 97**
- 47 alias for predefined struct type 97
- 48 mapping to system class **78**
- 49 members of a **78, 97**
- 50 struct type and **279**
- 51 string **101**
- 52 struct 95, 96. *See* struct
- 53 constructors in a **96**
- 54 predefined 97
- 55 unmanaged **334**
- 56 unsafe **331**
- 57 value 16, 95
- 58 constructor and **96**
- 59 conversion to/from a reference type **102**
- 60 sealed 96
- 61 struct **280**
- 62 value versus reference 95, 96
- 63 void* **334**
- 64 volatile **232**
- 65 Type 157
- 66 type expression *See* expression, type
- 67 TypeInitializationException
- 68 no matching catch clause and **318**
- 69 typeof 129, **157**
- 70 u integer literal suffix **62**
- 71 U integer literal suffix **62**
- 72 uint 18, 97, **98**. *See also* UInt32
- 73 UInt16 79, 97
- 74 members of **79**

1	UInt32.....	79, 97	38	using-statement	32, 207
2	members of	79	39	UTF-8.....	3
3	UInt64.....	79, 97	40	<value>	468
4	members of	79	41	value	130
5	ul integer literal suffix	62	42	default	107
6	uL integer literal suffix	62	43	value type	96
7	Ul integer literal suffix	62	44	enum member.....	308
8	UL integer literal suffix	62	45	Not-a-Number	<i>See</i> NaN
9	ulong.....	18, 97, 98. <i>See also</i> UInt64	46	reference type.....	<i>See</i> instance
10	unboxing	22, 102	47	set accessor and.....	37, 251, 263
11	unchecked	98	48	value expression.....	<i>See</i> expression, value. <i>See</i>
12	constant expression and.....	183	49	expression, value	
13	explicit numeric conversion and.....	122	50	value parameters	<i>See</i> parameter, value
14	integer addition and	166	51	ValueType.....	281
15	integer division and	164	52	variable.....	22, 105
16	integer subtraction and.....	168	53	definitely assigned.....	105, 108
17	multiplication and.....	163	54	exception	203
18	operator.....	158	55	catch without an	204
19	shift operations and.....	170	56	fixed	336
20	statement.....	32, 206	57	initially assigned	105, 108
21	unchecked operator versus.....	206	58	initially unassigned	105, 108, 109
22	unary minus and	161	59	instance	23, 105. <i>See</i> field, instance
23	#undef.....	68	60	definite assignment and.....	106
24	applying to undefined name	69	61	in a class	105
25	Unicode.....	18, 55	62	in a struct.....	106
26	char type and.....	98	63	initializer	271
27	string type and	101	64	life of an	105, 106
28	Unicode standard	3, 5	65	iteration	197
29	unsafe.....	27, 331	66	local.....	16, 22, 30, 107
30	unsafe code	8, 27, 331	67	declaration.....	76, 107, 189
31	stack allocation and	346	68	for and	196
32	unsafe context.....	331	69	declaration of multiple	189, 190
33	ushort	18, 97, 98. <i>See also</i> UInt16	70	definite assignment and.....	107, 189
34	using-directive	15, 49, 209, 210	71	scope of	189, 194
35	order of multiple	211, 212	72	movable.....	336
36	permitted location of a.....	210	73	reference.....	<i>See</i> reference
37	scope of a.....	211	74	static	23, 105. <i>See</i> field, static
			75	definite assignment and.....	105

1	variable expression	<i>See</i> expression, variable	14	unnecessary new usage	222, 293
2	variable reference	117	15	unreachable statement	186
3	versioning	50	16	user-defined.....	71
4	virtual.....	43, 243	17	while.....	31, 195
5	base access and.....	152	18	break and.....	195
6	void.....	129, 237	19	continue and	195
7	void*	334	20	reachability and	195
8	casting to/from a	334	21	white space.....	56, 58
9	volatile	232	22	XML.....	461
10	#warning	71	23	zero	
11	warning		24	negative	98
12	compile-time.....	8	25	positive	98
13	hiding an accessible name	88, 89, 221, 293			